TestRail

Best of the TestRail Quality Hub

Test First: TDD, BDD and Unit Testing

TestRail

It's well-known that the most costly defect to fix — in time, effort, and pain to the team — is one that makes it into production. Moving testing earlier in the development lifecycle helps to identify and resolve defects while they are still relatively inexpensive. This ebook comprises the best recent articles from the TestRail blog on three major approaches for early testing: unit testing, Test-Driven Development (TDD), and Behavior-Driven Development (BDD).

Contents

Test First: TDD, BDD and Unit Testing

Unit Testing, TDD, and BDD: An Introduction

Erik Dietrich, DaedTech LLC

Have you ever wondered about the differences between unit testing and TDD? Or TDD and Test-First? Or even the relationship of all of this to the agile methodologies? Are these kinds of tests manual or automated? And, for goodness' sake, what is it with all the "DD"-ending acronyms?

If the paragraph above accurately describes you and your relationship with these topics don't despair, because we've got your back. Today's post is your guide to the different testing techniques and methodologies available.

After reading this article, you will:

- understand what unit testing is about;
- know about TDD and BDD, understanding their place in the software development landscape and how they relate to each other; and
- learn about the tools and sources of information at your disposal.

Let's get started.

First Things First: Automated Software Testing Is a Thing

It may come as a surprise to some of you that I even felt the need to add such a section. If that's the case for you, feel free to skip it. Believe it or not, there are many developers for whom the notion of automated software testing is completely alien.

If you go to Wikipedia, you'll see that the topic of automated software testing is huge. Really, really huge. There are many different kinds of automated testing, each one of them catering to some specific need or even audience.

Nowadays, when people talk about automated testing (or simply testing), more often than not they mean unit testing, which many people consider to be the most important type.

Unit Testing: The Groundwork for All That's to Come

There are many possible and somewhat conflicting definitions for unit testing. I particularly like Roy Osherove's definition:

> *A unit test is an automated piece of code that invokes a unit of work in the system and then checks a single assumption about the behavior of that unit of work.*

Let's try to parse this sentence. Roy starts out by claiming that "a unit test is an automated piece of code." In other words, it's a **program**. A unit test is an automated test. It's a program that tests your program!

Let's move on. The definition goes on to state that a unit test "invokes a unit of work in the system." Let's not worry for now about what a unit of work is; instead, focus on the word "invokes." Our automated unit test invokes whatever it's testing: no human intervention needed.

Finally, we see that the unit test then "checks a single assumption about the behavior of that unit of work." Again, don't worry about what a unit of work is or isn't. Instead notice that, once again, no human intervention is needed, since the test itself checks its result.

What can we get from this definition? In short, a unit test is a program that:

- invokes some piece of code;
- compares the result of said invocation with some desired behavior; and
- accomplishes all of that without human intervention.

And how does all of that work in practice? In short, we can say that you write and run unit tests with the help of a test framework. In the Java world, for instance, the most well-known test framework is JUnit, while in .NET-land you'd probably use NUnit or xUnit.net.

And What About the Unit?

In the previous section, I kept telling you not to worry about what the unit (or "unit of work," as Roy Osherove puts it) means in the context of unit testing. Now it's time to worry. So...what is this unit thing?

Well...it's kind of hard to say. One of the most controversial things about the whole unit testing thing is the definition of a "unit" itself.

This is what Wikipedia has to say about the issue:

> *Intuitively, one can view a unit as the smallest testable part of an application. In procedural programming, a unit could be an entire module, but it is more commonly an individual function or procedure. In object-oriented programming, a unit is often an entire interface, such as a class, but could be an individual method.*

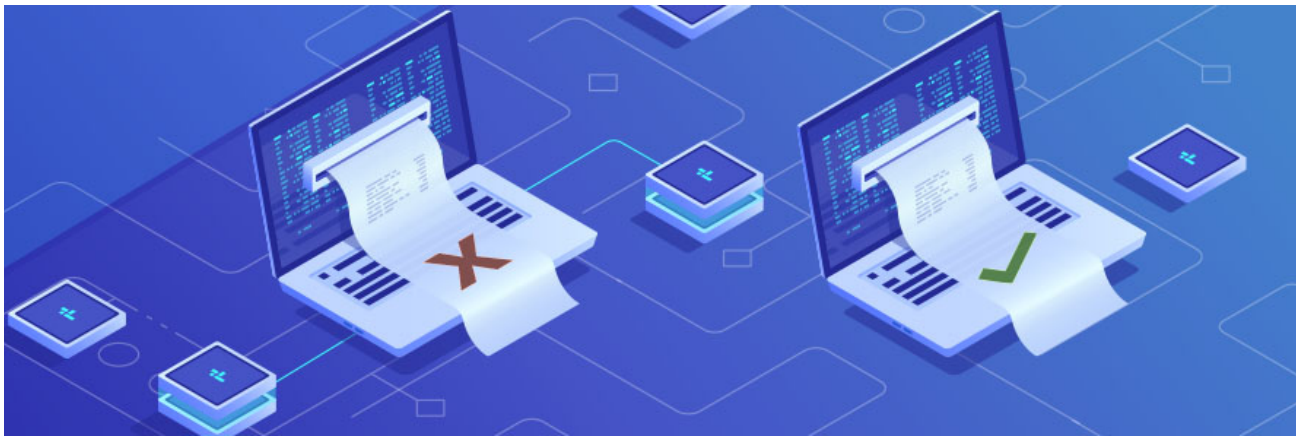
And what about Roy Osherove's definition? Let's see:

> *A unit of work is a single logical functional use case in the system that can be invoked by some public interface (in most cases). A unit of work can span a single method, a whole class or multiple classes working together to achieve one single logical purpose that can be verified.*

In my view, what all of those definitions have in common is that they're somewhat vague. It doesn't sound that useful to me to go to a beginner and say: "What is a unit? Easy: a unit is a class, except when it's just a method. Oh, I forgot to mention: it can also be a group of several classes."

For that reason, I tend to say that a unit is just a method of the class under test. It might be somewhat reductionist, maybe not totally correct, but it sure is pragmatic. And of course: your definition doesn't have to remain the same forever. It can—and should—evolve as you learn and gain more experience with all of this.

TDD: Unit Testing Driving You to Better Design



TDD means **Test Driven Development**. It's a software development methodology in which unit tests are used to drive the development of the application.

TDD was created/rediscovered by Kent Beck, who released Test Driven Development: By Example in 2002. TDD has gained a lot of popularity since then, in part due to being one of the key engineering practices of the Extreme Programming methodologies.

Great, but How Does TDD Work?

The process of applying TDD is itself very simple. You should write code following what's called the "red-green-refactor" cycle:

- Before starting to implement a feature, you should write a test for it.
- The test will obviously fail, since the thing being tested doesn't exist.
- You then proceed to write the minimum amount of code that will make the test pass. Cheating (i.e., taking a shortcut that doesn't really solve the problem but causes the test to pass) is not only allowed, but actively encouraged.
- As soon as the testing passes, it's time for refactoring (i.e., improving the code without changing its behavior).

Proponents of TDD claim that by following this process you'll achieve a simpler design, creating modules that are by definition low coupled and having more confidence to make changes to the code in the future, since you'll have an automated test suite covering all of the code.

BDD: Behavior-Driven Development

BDD means Behavior-Driven Development and was introduced in 2006 by Dan North. Dan states that BDD evolved as a response to TDD, an attempt to address issues he had when trying to apply the TDD process.

BDD emphasizes the need to include not only software developers but also non-technical people, such as business analysts, in the process of defining the tests. By the way, the term **test** itself isn't that welcome anymore. Under the light of BDD, we should think of **requirements**. Requirements should follow the template:

- **Given** some initial context (the givens),
- **When** an event occurs,
- **Then** ensure some outcomes.

By using a dedicated tool (such as Cucumber, JBehave, or Specflow), it's possible to turn requirements written in this format to a test skeleton.

And How Do Unit Testing, TDD, and BDD Relate to Each Other?

How do all of these pieces fit in the puzzle? What are the relationships between them?

Unit testing is a type of automated testing. You can write unit tests without using TDD or BDD, just by writing the tests after the production code.

TDD is a software development methodology, in which the developer writes code in very short cycles, always starting with a failing test.

BDD can be thought of as a "flavor" of TDD, in which the application's development is driven by its behavior, described in the forms of human-readable requirements that must be later converted to executable tests.

The Verdict?

All three of the techniques (but especially unit testing and TDD) presented in today's post remain more or less controversial to this day. See some examples:

- Test Introduced Damage
- TDD? Waste of time!
- Write Tests. Not Too Many. Mostly integration.

My personal opinion on these techniques has changed over the years but my current take is: write automated tests for your code. Mostly unit tests, but also other types (while being aware that there might be exceptions to that rule). The process that you use in order to write said tests shouldn't matter that much.

Are TDD and BDD worth your time? Definitely. Do they also provide challenges and have their own shortcomings and limitations? Of course. There's no silver bullet in our industry. The only way is to read a lot and practice a lot in the hopes that, when the time comes, you'll be able to make informed decisions.

Test First: TDD, BDD and Unit Testing

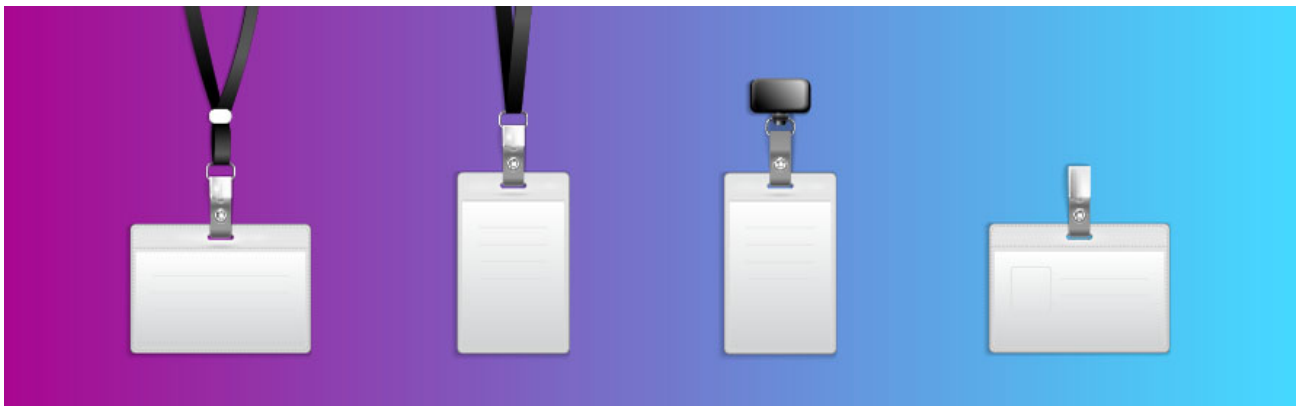
5 Elements of Good, Maintainable Unit Tests

Erick Dietrich, DaedTech LLC

While specific methodologies such as TDD still attract a lot of controversy, I think it's fair to say that unit testing itself is a lot closer to a consensus. We're clearly not there yet, but we're getting closer and closer.

With that in mind, what can a software developer who is still a unit testing novice do to ensure their unit tests are good, maintainable, and reliable? If you match the description above, then today's post is for you. We'll show you five simple tips you can follow to improve the quality of your tests.

1 They Have Good Names



Some people say you should think of a unit test not just as a test but also as a kind of specification for the software. I wholeheartedly agree with this statement and one of its most important consequences is that a good, maintainable unit test should be very easy to read.

And that starts with its name. The name of the test should be written in a way that is extremely easy for the developer to quickly understand what exactly went wrong or when the test fails. OK, we agree that naming is important. But how do we go about that, since there are so many different conventions?

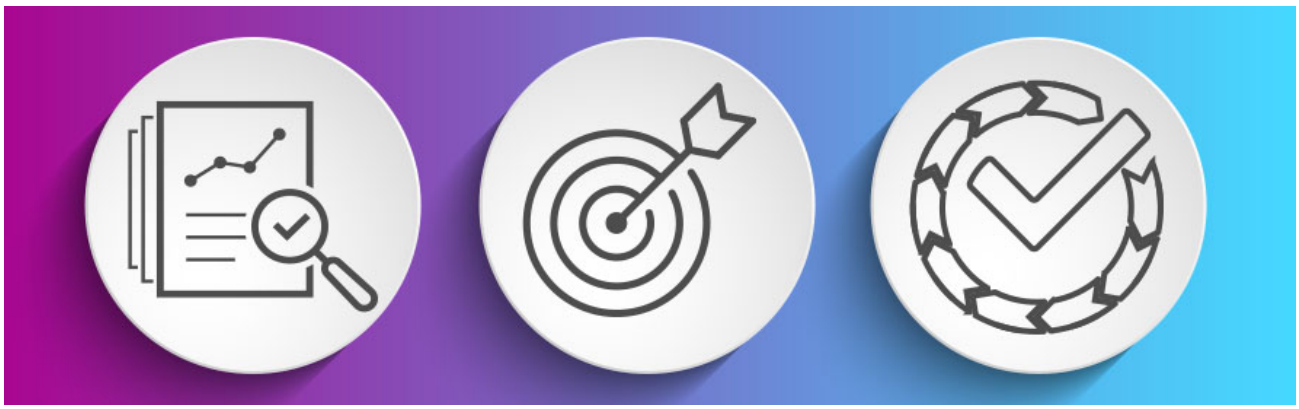
I particularly like Roy Osherove's convention which follows the "MethodUnderTest_Scenario_DesiredOutcome" format. For a quick example, let's consider the String Calculator Kata, also from Osherove.

One of the Kata's requirements states that the "Add" method should return 0 if you provide it an empty string. Using the above mentioned convention, you could name the test "Add_WhenGivenEmptyString_ReturnsZero" or something even simpler.

One of the possible drawbacks for this convention is that it will require you to rename the tests when you rename a production method. It's a valid criticism, but you should also have in mind that constantly renaming methods in your public API isn't such a good thing to do.

Anyway, this is just one of the many possible naming conventions out there. You could use it as is, maybe tweak it a bit to fit your needs, or even use another convention entirely. What really matters is that you name your tests in such a way that it's as easy as possible to understand what went wrong when they break.

2 They Follow the "Arrange, Act, Assert" Structure



The "Arrange, Act, Assert" pattern is a widely known way to structure the code in a unit test. It consists of breaking up the code inside a unit test into three clearly divided groups, each one representing a phase in the test:

Arrange. In this phase, you do whatever preparation you need in order to run your test. Instantiation of the System Under Test will usually happen in this phase.

Act. The name pretty much says it all. In this phase, you generally do whatever action you want to test.

Assert. Finally, it's time to verify if we've got the desired results.

What does that look like in practice? Basically, write your tests in such a way that the phases are clearly recognizable and then **respect each part**. Don't do assertions on the Act phase, don't arrange in the Assert phase, and so on.

Some people will even insist that you add comments indicating each phase. I don't think that's entirely necessary, but if it makes things easier for you, then go for it! The following code shows an example of unit testing with comments delimiting each phase:

```
[Test]
public void Construction_Properties()
{
// arrange
LocalDate start = new LocalDate(2000, 1, 1);
LocalDate end = new LocalDate(2001, 6, 19);

// act
var interval = new DateInterval(start, end);

// assert
Assert.AreEqual(start, interval.Start);
Assert.AreEqual(end, interval.End);
}
```

The example above was taken from the Noda Time project and edited to add comments. Whether you use comments, blank lines, or even another type of formatting strategy, the only thing that really matters is that the phases are easily recognizable.

3 They Don't Duplicate Production Code

You know what's worse than having no tests at all? To have tests that lie, give you a false sense of security, or most of all— to have tests that pass when they should be breaking. And a great way to achieve this awful outcome is to duplicate logic from the production code in the tests.

What does that mean? To answer that, let's insist on using the String Calculator Kata. Let's say you have a test like the following one:

```
[TestCase("1, 2", 3)]
[TestCase("5, 2", 7)]
[TestCase("4, 9", 13)]
public void Add_PassingTwoNumbers_ReturnsTheirSum(string numbers, int
expectedResult)
{
Assert.AreEqual(expectedResult, StringCalculator.Add(numbers,
expectedResult));
}
```

The test passes and all is fine with the world. But you're a developer, which means that you're pretty much guaranteed to, eventually, have the following thought: "It's kind of ugly to hardcode the expected values like this. Maybe there's a better way?"

And then you create something like this:

```
[TestCase("1, 2")]
[TestCase("5, 2")]
[TestCase("4, 9")]
public void Add_PassingTwoNumbers_ReturnsTheirSum(string numbers)
{
var expectedResult = numbers.Split(',').Select(int.Parse).Sum();
Assert.AreEqual(expectedResult, StringCalculator.Add(numbers,
expectedResult));
}
```

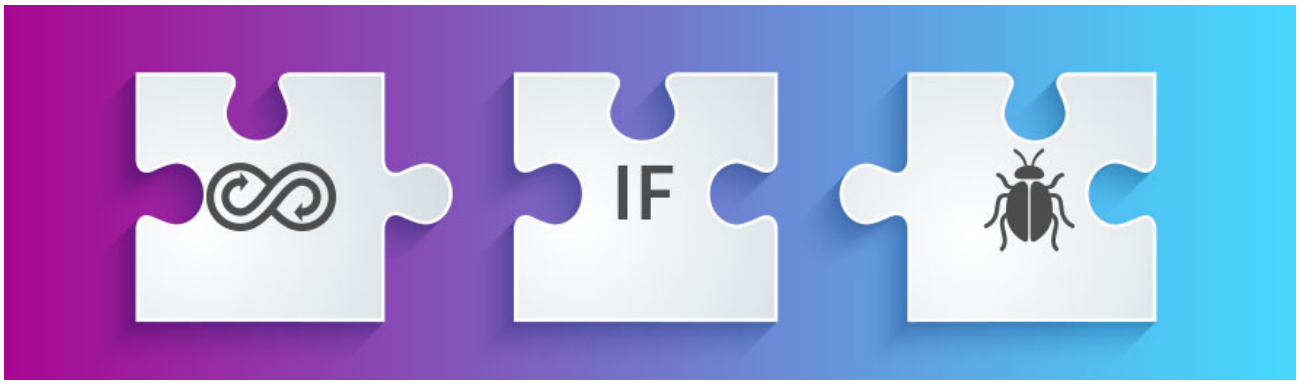
What's wrong with the code above? As far as I can tell, nothing. When I run the test in my machine, it works just fine. So what's the matter?

By trying to automate the generation of the expected value, you're potentially duplicating the production code in the test (in this particular case, I'm **definitely** doing it since I just copied the code from the "Add" method). But again: what's the matter?

The matter is that this is very dangerous. In the eventuality that the production code is wrong, the test would also be wrong. Not only that, though: of all the infinite ways it could be wrong, it'll be wrong in the exact way that will make the test pass.

This is particularly dangerous when doing TDD, since both the production and test code are developed concomitantly and generally by the same person (if they're not pair-programming). And then you'll have one of the worst possible scenarios: **the code is wrong but the test passes,** which can cause you to deploy buggy software to your users, but also can undermine the team confidence in the discipline of unit testing itself.

4 They Don't Contain Loops and/or Conditional Logic



This is related and sort of a continuation of the previous point. Your unit tests shouldn't contain loops or decision structures.

There are two reasons for this:

- First, tests with conditionals and loops become harder to read (which calls back to the first point.
- But more importantly, tests containing these constructs might contain bugs themselves.

Think of it this way: how can you trust a test that's so complicated to the point of needing to be tested itself? **You can't.**

OK, I swear I can even hear you arguing: "But man, I really need to iterate over this list in my test because yada yada."

In this case, here's what you should do: move the code that performs the looping/conditional logic for a dedicated, utility class in your testing assembly. From your main test, you call the utility class.

And of course: write tests for the utility class itself!

5 They Are Truly Independent

Your tests should be truly independent. “Independent from what?” you may ask. Everything, everyone. What do I mean by this?

For starters, each unit test should be independent from the other unit tests. If your test suite requires that the tests should be run in a certain order, then they’re not really unit tests. If you have tests A, B, and C, then each one of the following scenarios should result in the tests passing:

- You run only A.
- You run only B.
- You run only C.
- You run all of them, from C to A.
- You run all of them, from A to C.
- You run all of them, 100 times each.
- Any other combination you can think of.

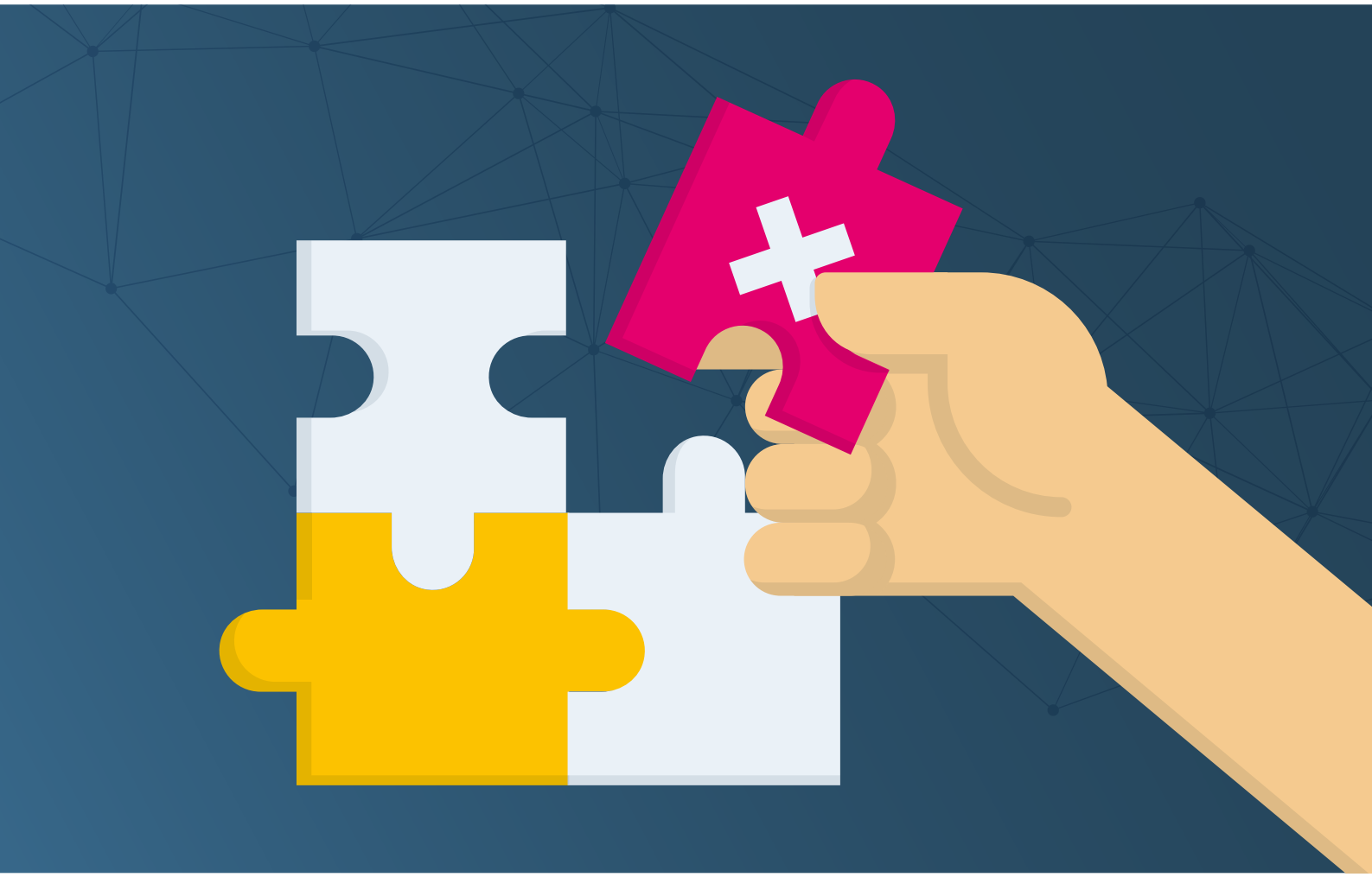
The tests should also be totally independent from the outside world. They shouldn’t rely on a database or some file. Nor should they rely on specific things about the context of the machine they’re being executed on, such as the system’s clock or the system’s current culture.

Is That All There Is to It?

Is that all there is to writing good unit tests? Of course not. Entire books have been written about the subject. There are plenty of courses online—both paid and for free—offering to teach you strategies and techniques to master unit testing. It's obvious that I could never exhaust the subject in a humble blog post.

That being said, I do believe that the tips we've provided are a great starting point. With this framework we just gave you, you now have a very solid foundation on which to build your knowledge of unit testing techniques.

As they say, "Practice makes perfect." Continue working, never stop practicing. Happy testing!



Test First: TDD, BDD, and Unit Testing

The Seven Sins of Unit Testing

Erik Dietrich, DaedTech LLC

The debate over the merits of unit testing has raged for the better part of two decades. But it's no stalemate. The tide has moved steadily toward an industry that more or less, for the most part, agrees that unit testing is a good thing to do.

By and large, this is a good thing for the state of the industry and the state of the art in software. More developers writing more unit tests means catching regression defects earlier and less fear when changing existing code. But it also means more developers diving into unit testing, not fully comfortable with how to do it just yet. And that can lead to mistakes and headaches.

So today, I'm going to talk about mistakes that developers often make when writing unit tests. You'll typically see these so-called sins of unit testing when people are first start unit testing. But if they're not careful, it's more than just a temporary growing pain. These problematic tests become baked in to the codebase and stick around for the long haul. Let's look at what to avoid.

1 Slow Running Tests

I'll start with something that may seem like a bit of a nitpick. But it's actually foundational. You absolutely do not want to tolerate slow-running tests in your unit test suite.

What's so bad about slow running tests?

It's not some nebulous notion of performance, nor is it the wasted developer time, per se. Oh, those things matter. But the foundational problem is that long-running tests bore developers. And bored developers, looking to write code and be productive, remove the boring obstacle. In this case, they remove it by **not running the test suite.**

If you have slow unit tests, you have a test suite that will languish, unused. You might as well not bother. You can certainly have long running tests in your test portfolio. But keep them separate from your developers' unit test suite, which the team should run constantly.

2 Lots of Assertions

I made this mistake myself, years ago, when new to unit testing. I wrote tests with lots of assertions.

```
[TestMethod]
public void Test_All_The_Things()
{
var customerProcessor = new CustomerProcessor();

customerProcessor.Initialize();

Assert.IsTrue(customerProcessor.HasCustomers);
Assert.AreEqual(12, customerProcessor.CustomerCount);

Assert.AreEqual("John", customerProcessor.getFirstCustomer().Name);
Assert.AreEqual("Smith"), customerProcessor.getFirstCustomer().Name);

Assert.IsFalse(customerProcessor.HasInvalidEntries);
}
```

If one assertion is good, more are better, right? You want to make sure the customer processor behaves correctly. Right?

Well, yes, you do. But not like this. To understand why, ask yourself this. If you saw on a unit test report that "Test_All_The_Things" had failed, would have any idea what the problem was, at a glance? Which assertion failed? Why?

When a unit test breaks, it provides you with a warning — the equivalent of your car giving you a warning on your dashboard. Do you want a warning that says, "low tire pressure" or "low battery" or would you rather have one that says, "something is wrong somewhere?" Probably the former. The same logic applies in your test suite. Each test should tell you something specific and detailed about what's going on with your codebase.

3 Peering into Privates

One of the most common questions I hear among newbies to unit testing is something like "how do I test private methods?" They're usually initially dumbfounded by my response. "You don't."

Unit tests are meant to serve as a way to exercise your codebase's public API. If you want to test the functionality of a class's private methods, then you do so indirectly by testing the public methods that call them. If this is really hard, it's a good sign that your code is too monolithic (e.g. lots of iceberg classes) and that you should extract some of this functionality to a separate class.

In many languages, you can use constructs like reflection to "cheat" and access methods labeled as private. Don't do this. You'll break encapsulation and create really brittle unit tests. Instead, extract the private implementation to a new class with a public interface that the existing class uses privately. It's the best of all worlds — you retain encapsulation and have an easier time testing.

4 Testing Externalities

Remember that the first sin involved writing long-running unit tests? One surefire way to write unit tests that take forever to run is to write unit tests that do things like writing files to disk or pulling information out of databases.

So avoid expensive use of externalities because it slows down your test. But also avoid it because, when you do this, you're not actually writing unit tests.

Unit tests are focused, fast running checks that isolate your code and assert how it should behave. You're checking things like "if I feed the add(int, int) method 2 and 2, does it return 4?" That's the scope of a unit test.

When you're executing code that calls web services, writes things to disk, or pulls things from a database, you're actually writing integration or end-to-end tests that, by definition, are not testing things in isolation. You're involving external systems which means that your "unit" tests can fail for environmental reasons that have nothing to do with your code.

You can avoid this particular sin by learning more about the unit testing technique of **mocking**.

5 Excessive Setup

If you find yourself writing a unit test that seems particularly laborious, you should probably stop and do a quick sanity check. Do you have dozens of lines of code instantiating things, passing them to other things, mocking all kinds of objects, and just generally doing a lot of busy work? If so, realize that setup is excessive.

Any number of things can create a situation with excessive setup. It might be a lack of familiarity with test writing and mocking, creating inefficiency in the tests. Or it can be a simple case of highly coupled design. If you have to set six different global variables in a specific sequence in order to test the code you want to test, you should revisit how your production code works.

But whatever the case, try to avoid excessive setup. Make the setup more efficient and/or improve the production code. Because tests with lots of setup are extremely brittle and they make maintaining the unit tests suite an onerous chore. They make it the kind of onerous chore that the team will simply stop doing.

6 Daisy Chaining Tests

This sin is more subtle, but also crucial to understand. Your unit tests should each be self-contained and possible to run in isolation. If each one were the only test in the entire suite, it should work just as faithfully as it does in mixed in with the others. Never make it necessary to run your unit tests in a certain order.

```
[TestMethod]
public void Test1()
{
GlobalVariables.IsProcessorInitialized = CustomerProcessor.Instance.
Initialize();

Assert.IsTrue(customerProcessor.HasCustomers);
}

[TestMethod]
public void Test2()
{
var firstName = CustomerProcessor.Instance.GetFirstCustomer().FirstName;

Assert.AreEqual("John", firstName);
}
```

Here we have global state in the form of a singleton, and that singleton apparently requires initializing beyond just instantiation. Test1 initializes the processor, and then Test2 assumes that the processor is initialized and goes on to test other concerns.

Do not do this!

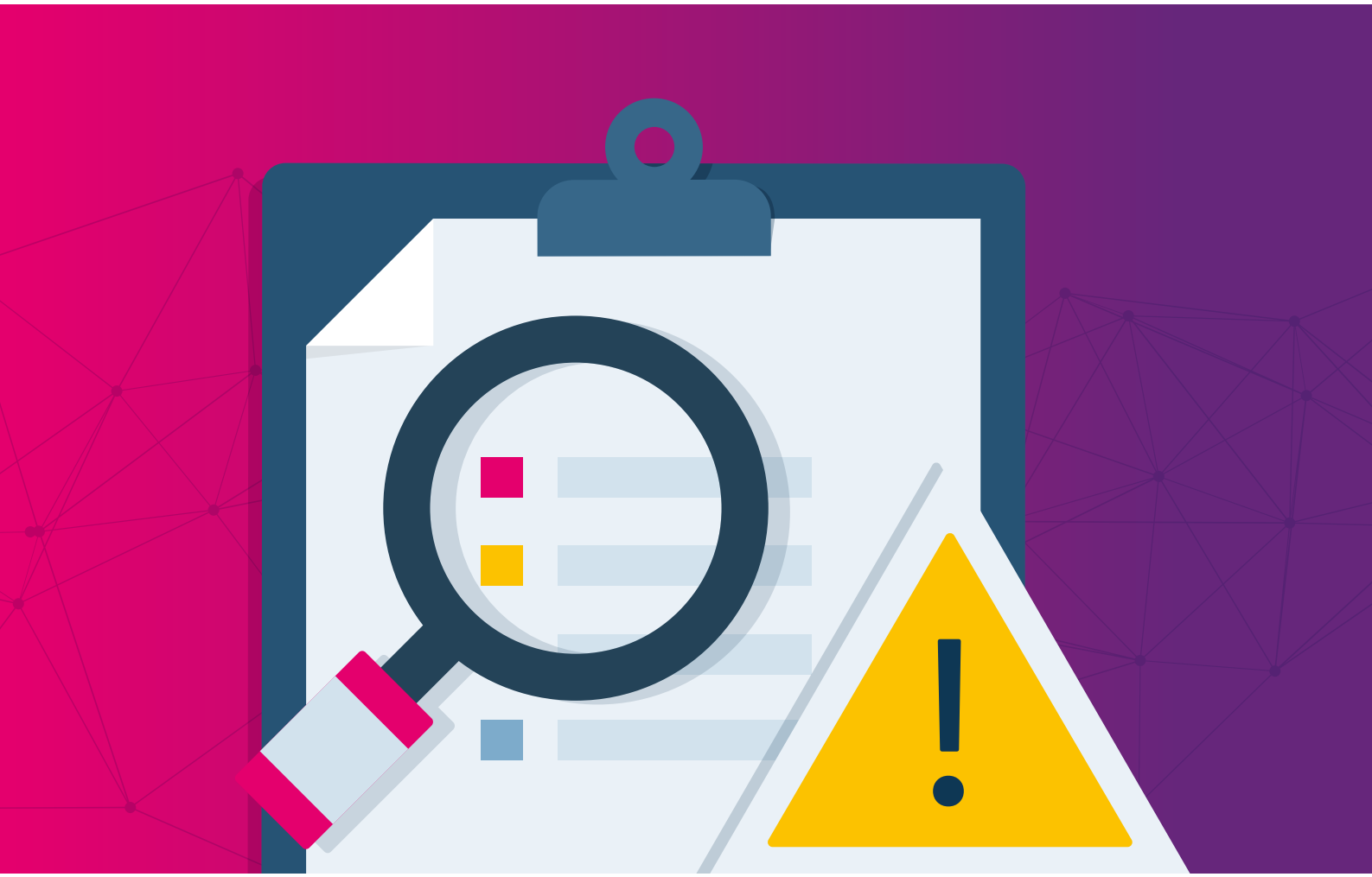
Many test runners make no guarantees about the order in which your tests run, and will even run them in parallel. And, even if you can force ordering with your runner, this may be a configurable setting that others do not enable or that does not run on the build.

This type of test leads to test suite nightmares — scenarios in which your tests will fail intermittently and seemingly randomly. You'll then hear developers say, "oh, the test failed on the build machine? Just try it again or ignore it." And this defeats the whole purpose of having a test suite to warn you of trouble.

7 Test Code as Second Class Citizen

I'll close with perhaps a more philosophical than tangible 7th sin. This sin involves treating your test code as a second class citizen. You'll know this is happening when team members say things like, "whatever, it's just test code, so who cares if we copy and paste?"

Treat your test code with just as much care as your production code. You're going to need to maintain both sets of code over the long haul, and both are critical to ensuring that your code behaves properly in production. So don't skimp on either.



4 Outside-in Signs That You Don't Have Sufficient Unit Testing

Erik Dietrich, DaedTech LLC

Starting unit testing can be tough. Many questions, misconceptions, and points of confusion torment the beginner. One of the most common of such doubts revolves around amount. What's the correct amount of testing I need? How much is enough? Should I aim for 100% coverage?

In this article, we'll help you answer some of these questions, but not by telling you how much is enough. Instead, we're sort of doing the opposite. We'll give you tools to identify how few tests is too few.

1 Test Coverage Is Low

The first and most visible sign of a lack of unit testing is, unsurprisingly, low test coverage. Test coverage is a topic that can spark fierce debates. Some developers will fight religiously to defend their point of view on the issue. The issue being: is test coverage really that useful of a metric?

I'm not settling that matter in this post, nor do I intend to. One thing for me is obvious, though. While we can't seem to agree whether 100% coverage is a good thing or not, we can agree that shallow coverage is a bad sign.

How can developers gain confidence in the test suite if it only covers a narrow portion of the code base? The answer is that they can't. When a developer sees the green bar on their screen, they can be pretty confident that their code is correct. If the number of unit tests isn't high enough, having such a degree of confidence is just wishful thinking.

2 Test Coverage Isn't Increasing

Picture this: you work at a small-sized software shop. There are about six or seven developers currently on the team. There's also a QA department, comprised of a test analyst and two interns.

Some unspecified time ago, management heard something about unit testing and brought in an external consultant to provide training for the team.

Since then, the developers have been adding unit tests to the code base with varying degrees of dedication and success. Some of the developers on the team are really into it, others less so, and a few are openly skeptical about the whole thing.

You, being an advocate for unit testing, know you and your colleagues are lucky to at least have management on board with this (since it was their initiative). Some developers in other companies aren't so fortunate. But here's the catch. Even though management officially supports the unit testing initiative and have also put money into it, in practice, they only pay lip service to the importance and benefits of testing. When project deadlines start looming, managers pressure developers into skipping unit testing in favor of writing production code.

And what about the QA department? Well, they work around the clock, finding and reporting bugs every single day. And that's excellent work because those bugs won't affect customers. But you know what? Just about everyone knows that writing a new unit test every time someone files a new bug is a widely accepted best practice. Yet this doesn't seem to be happening.

If bugs are continuously being found and reported, test coverage should steadily increase. When that doesn't happen, it's a powerful indicator that your codebase needs more tests.

3 Existing Tests Rarely Fail

A telltale sign of insufficient unit testing is when developers seldom experience a test failing.

Don't get me wrong. Your tests shouldn't fail all the time. If they do, you might have problems with the test suite itself. Maybe the tests depend on implementation details of the production code. Perhaps the tests make unnecessary assertions (e.g., they expected a specific exception message, but someone found and fixed a typo on said message, causing the tests targeting it fail).

Or maybe the tests are rightly failing, which means the developers are introducing errors at an alarming rate (which should scare you, but you should also be relieved that you have unit tests in the first place).

But let's get back on topic. Now that it's clear that I'm not advocating for an incredible amount of test failures let's address the opposite extreme. A test suite that never fails can be just as bad as a test suite that always fails—just in a different way. Unit tests are supposed to help developers gain confidence in their work by catching their errors. But if your test suite never actually catches any errors, what good is it?

Reasons Why Your Tests May Be Failing by Not Failing

OK, we've just established that an error-catching, confidence-boosting mechanism that fails to catch errors and therefore to boost confidence is pretty much useless. The question then becomes why? Why doesn't the test suite catch bugs more often?

One possible answer is that the tests just aren't right. After all, test code is still code, and all code is prone to bugs. So it shouldn't come as a surprise that unit tests can contain bugs themselves. Fortunately, there are ways to counter that. You can have a second pair of eyes review every code that makes it to production, either in the form of pair-programming or a more traditional code review practice. You should employ some workflow on which you watch a test fail—in an expected way, that is—before it passes. Test-driven development (TDD) is an example of such a workflow.

Finally, you can also employ a technique called mutation testing. Mutation testing refers to a process in which an automated tool deliberately introduces defects throughout a codebase and then runs the test suite. Each defect introduced is called a mutation. If at least one unit test fails after the introduction of a mutation, we say that mutation was killed. If not, then the mutation survives.

But if you're already doing all of the above and you're sure that your tests are as good as they can be, and yet, they rarely fail...then something must be wrong. (I mean, maybe, just maybe, your tests aren't failing because your developers are so darn good and write perfect code pretty much every time. I find this highly unlikely though.)

That being said, I can only think of one solution left for this puzzle. If you're fairly confident that your tests are of good quality, and you have evidence that new bugs continue to be introduced in the codebase, but your tests refuse to fail, then the only logical conclusion is that you don't have enough tests.

It's just a matter of probability. The smaller the portion of the codebase protected by tests is, the less likely it is that the error you've just introduced will fall on that covered area.

4 When Tests Fail, It Isn't Due to a Bug

Here's another telling sign that you don't have enough unit testing. If, besides rarely failing, when tests do fail, it's more often than not due to something other than a bug in the production code.

Ideally, the reason a unit test fails is due to a bug in the production code. This is the very reason unit tests exist, after all. Sadly, in the real world, several reasons may cause a test to fail. To cite a few:

- **A bug in the test itself.** In the previous section, we talked about some techniques and tools to prevent this from happening, but there is no silver bullet.
- **Changes in the public API of the system under test.** How big of a deal it is to break changes in a public interface depends on the type of software you're building, but stability is generally a good and desirable thing.
- **Changes in the implementation of the system under tests.** On the other hand, tests failing due to internal changes in the system under test? That's a bad sign.

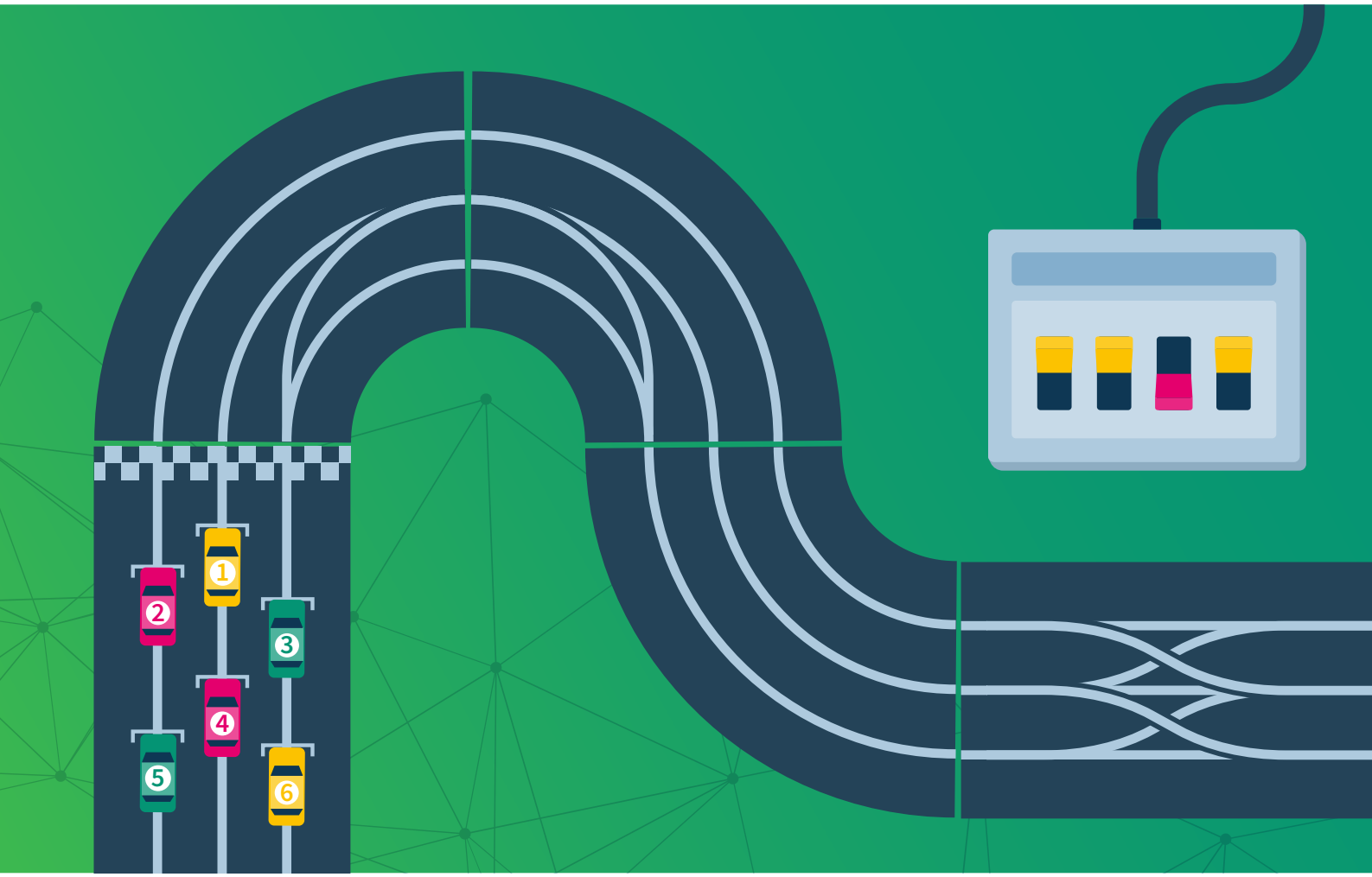
- **Miscellaneous reasons**, such as some problem with the CI system or server.

Here's the thing: **a test failing due to the reasons above should be the exception, not the rule.** Usually (and ideally), a test should fail due to an error in production. If you have tests that rarely fail and for the wrong reason when they do, that's not good at all.

Where There's Smoke, There's Fire

In this article, we've given you four signs you can use to identify when you don't have enough testing for your application. Don't jump to conclusions though. Use these signs as a starting point, the way doctors use a patient's symptoms to identify and treat the cause. Then, if it seems like you're on the right track, proceed to more testing.

And remember: the most crucial sign—both a symptom and cause—of a low number of tests is the lack of enthusiasm among developers. If you fail to create a strong unit testing culture in the development team, no amount of techniques or tools will perform miracles.



Find the Right TDD Approach for your Testing Situation

Jeff Langr, Langr Software Solutions, Inc.

While the test-driven development (TDD) cycle is simple — write a test, get it to pass, refactor — developers have found numerous ways to tweak the programming technique. In other words, no one true way to practice TDD exists. That means your oddball approach to TDD is probably OK, but it also means you can find a lot to learn from exploring some of these TDD "alternatives."

One Practice, Multiple Definitions

I described the TDD cycle simply as "write a test, get it to pass, refactor." You might have also heard the mantra "red, green, refactor" as a summary of this cycle.

This simple description might be enough to help you hit the ground running, but here's a more involved description of what doing TDD really means:

You write a unit test to describe a small bit of behavior that does not yet exist. The test consists of statements that first put the system under test into a known state, then exercise the desired behavior. The unit test needs at least one assertion — a statement that verifies whether or not some expected condition holds true

You run the unit test using a tool specific to your programming language. The tool will tell you that the test passed if all its assertions held true, or it will report that the test failed. With TDD, we want to ensure that the test failed

You write the minimal amount of code needed to make the test pass

Once you get the test to pass, you clean up any deficiencies in the code — things that will make it hard to understand and maintain in the future

Robert ("Uncle Bob") Martin presents TDD by specifying three rules you must follow:

1. Write no production code unless it is to make a failing unit test pass.
2. Write no more of a unit test than is needed to fail. Compilation failures count as failures.
3. Write no more production code than is needed to pass the one failing unit test.

So, what's the difference? For one, Uncle Bob's second rule, which implies that you must stop writing the test as soon as you receive compilation failures, is considerably more prescriptive about how to write a test — let's call this "incremental test writing." The stepwise description of TDD does not delve into an approach for writing the unit test, leaving that choice up to you.

I've done a mixture of both incremental test writing and wholesale "just-slam-the-whole-thing-out" test writing, and I found each to be useful. Often it's easiest for me to follow a stream of consciousness and flesh out an entire test. But I believe Uncle Bob includes the incremental-test-writing rule because there's value in taking smaller steps that provide feedback sooner.

Your language of choice might help you decide which approach works best for you. If you're in C++, where one compilation error triggers dozens more, it might be most effective to do incremental test writing. As soon as you receive a compilation error (nowadays indicated dynamically in a good IDE without the need for an explicit compile step), fix it. Taking such small steps will help you better correlate a given compilation error to its cause.

As with all the alternative approaches that follow, I highly recommend experimenting with this form of incremental test writing. You might find the results illuminating enough to improve your practice of TDD.

One marked difference between the two descriptions of TDD is that the three rules don't mention refactoring. The rules don't say not to refactor, either, and I'm sure Uncle Bob believes it's critical to success. Still, I prefer the stepwise description and its explicit inclusion of the refactoring step, because I believe the ability to continually address code cleanliness through refactoring is the best reason to adopt TDD.

Assert First

In the book "Test-Driven Development: By Example," Kent Beck tells us to try writing the assertions first. This prescriptive suggestion, which has you essentially working backward, can help you think more about the outcome (the "what") rather than the implementation details (the "how").

During my very long history with TDD, I've grown too accustomed to not writing the assertions first — in other words, I write the test more or less top to bottom. But occasionally writing assertions first makes the most sense for the challenge at hand, most typically when I have a lot of unanswered questions about the codebase and how the new behavior will impact it.

One side effect that assertion-first approach seems to have is that the focus on outcome means I often end up using programming by intention: Because I don't yet know what the details need to be in the rest of the test, I start by writing the name of a yet-to-be-implemented helper method. My test rises in its level of abstraction — the focus is on what to do, less how to do it. I then flesh out the helper methods.

Here's a unit test that I coded top to bottom many years ago:

```
@Test
public void returnsHoldingToBranchOnCheckIn() {
service.checkOut(patronId, bookHoldingBarcode, new Date());

service.checkIn(bookHoldingBarcode, DateUtil.tomorrow(), branchScanCode);

Holding holding = service.find(bookHoldingBarcode);
assertTrue(holding.isAvailable());
assertThat(holding.getBranch().getScanCode(), equalTo(branchScanCode));
}
```

The test reads procedurally well, but the three lines of assertion are a bit of a mess. I got there by knowing I could retrieve a holding using the service, then asking some questions of it. With an assert-first approach, I would start with a single line to express the expected outcome:

```
assertThat(service.find(bookHoldingBarcode), is(availableAt(branchScanCode)));
```

The matcher method `availableAt` doesn't exist yet. At this point in writing the test, I don't yet know exactly the steps I'll use to implement it; nor do I care. Meanwhile, I've been able to craft a very literary assertion that declares the outcome rather than making the reader work stepwise through it.

TDD itself is a programming-by-intention technique.

A Single Assert Per Test

Dave Astels promoted the controversial notion of one assert per test almost 15 years ago. His advice isn't quite as controversial when it comes to preventing run-on tests that work through multiple cases, as shown below:

```
@Test
public void bankAccount() {
var account = new BankAccount();

// balance is zero when created
assertThat(account.balance(), is(equalTo(0)));

// deposits
account.deposit(100);
assertThat(account.balance(), is(equalTo(100)));
```

```
account.deposit(200);
assertThat(account.balance(), is(equalTo(300)));

// withdrawals
account.withdraw(50);
assertThat(account.balance(), is(equalTo(250)));

// ...
}
```

It's a little easier to slap together a run-on test. Often the various cases (indicated by the comments in the above example) depend on a bit of setup context. Creating a separate test method for each case would require some redundancy in the setup for each individual case, and perhaps that's why some people balk at the idea.

It's easy to factor out such redundancies, however, using setup hooks and helper methods. Some folks perhaps are concerned about the execution redundancy, but if we're writing isolated unit tests that have no dependencies on slow collaborators, adding new sub-millisecond tests is a non-issue.

Here's what a single-assert-per-test approach looks like:

```
private BankAccount account;
@Before
public void createAccount() {
account = new BankAccount();
}

@Test
public void hasZeroBalanceWhenCreated() {
assertThat(account.balance(), is(equalTo(0)));
}

@Test
public void increasesBalanceOnDeposit() {
account.deposit(100);

account.deposit(200);

assertThat(account.balance(), is(equalTo(300)));
}
```

```
@Test
public void decreasesBalanceOnWithdraw() {
account.deposit(300);

account.withdraw(50);

assertThat(account.balance(), is(equalTo(250)));
}
```

Each test describes one behavior, which provides a few advantages:

The isolated nature of each test can make it much easier for readers to understand the intended behavior

The test name concisely summarizes the behavior, making it possible for the list of test names to help maintainers understand where their changes need to go

On test failure, it's much easier to uncover the source of the failure — side effect errors created by one case do not generate errors in subsequent cases

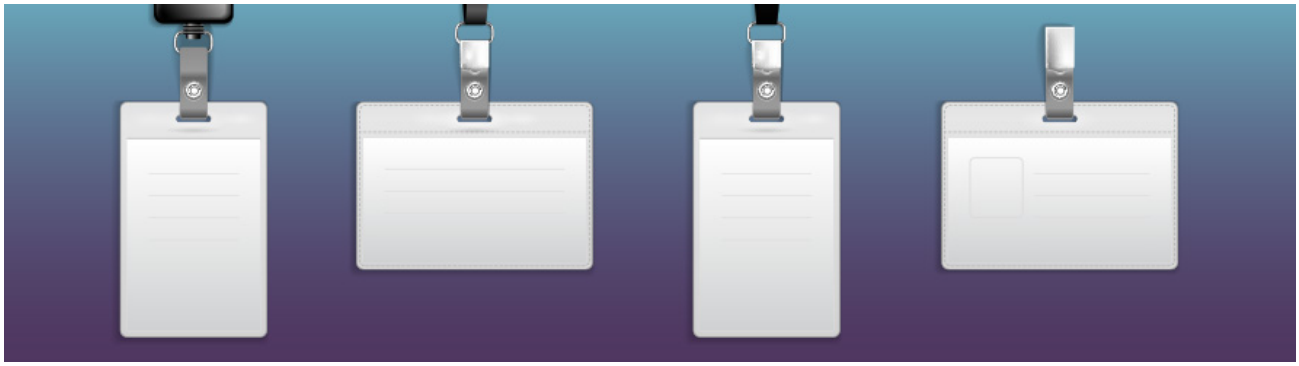
Does "one assert per test" always make sense? What if you're verifying that a dozen fields were shuttled over from a cursor into a domain object?

Perhaps it's better to think of "one assert per test" as "one behavior per test." You might consider that copying a bunch of related columns into associated fields is a singular behavior. How do you know?

My take: **Start with a single assert**. If you can't think of a meaningful way to name the next test with a unique behavioral description, you're probably OK with combining the asserts into a single test. Otherwise, stick with a single assert per test.

Always consider that odd coding challenges like this one might represent a smell. Does the compulsion to combine multiple asserts into a single test indicate something suspicious about the design of your production code? In the case of data shuttling, a data dictionary approach might be the right cure that simplifies your system overall and allows you to stick to one assert per test.

Test Naming



In TDD, there are various approaches to naming your tests. You might use the form `DoesSomethingWhenSomeContextExists`. You might also go with `WhenSomeContextExistsSomethingHappens`, or you might even use `GivenSomeContextWhenSomeEventOccursThenSomeConditionHoldsTrue.`

For a few years, I've promoted an alternative: I name my test classes or fixtures starting with the article "A" or "An." The test class combined with each test name completes a sentence:

```
TEST_F(AnAutomobileWithEngineStarted, HasLowIdleSpeed) {
/* … */
}
```

Or:

```
class ACheckedOutHolding {
[Test] public void IsAvailableAfterReturnToBranch()
{
/* … */
}
```

Naming is one of the most important things you do! Choose whichever naming form is most appealing to you. It won't matter as long as you're consistent across the tests and the test names clearly describe intended behavior.

Nameless Tests

When test driving, there are many ways to skin a cat. I rarely believe there is an absolutely right one way to do any given thing. That means it's up to you and your team to discuss and settle on a technique that works best for your situation.

With most of the above choices I've described, I settled on one approach because I found value through employing it. My choice doesn't imply that the other approaches are wrong; if your team takes an alternate approach, I'm happy to go along with it. Only in rare cases have I recoiled in horror upon seeing an alternate approach.

All of my tests are named. That's to support their value of documentation. If you're going to invest this much effort in writing tests, they should pay off in multiple ways. Describing the intended behaviors of the system is one such way.

I've heard at least one person espouse the notion of nameless test cases, however. Their contention:

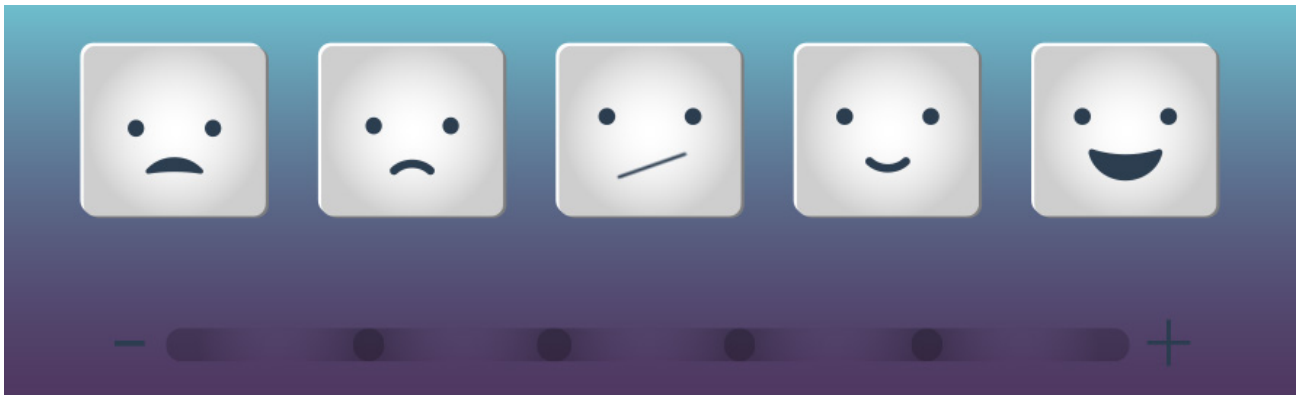
- Test names are comments, and as such could be lies that inaccurately describe the test code contained within
- Tests should be written as highly readable examples, meaning they should not need a summary

I played with this idea of nameless tests with an open mind for a day or two. As with any of the earlier variants I described, I always recommend experimenting with the ones you're not comfortable with before making a decision.

In this case, I firmly came down on the side of "no way." First, I don't want to waste time reading through dozens of lines of examples in order to find the ones that pertain to what I need to change in the code. Sub-section headings exist in textbooks for a very

similar reason. Second, an example of behavior, no matter how well you name the variables and functions and variables it employs, doesn't always concisely express the real intent. Nameless tests are an interesting idea, but one I think is ultimately damaging. I tried it fairly, I didn't care for it, and I won't employ or recommend it.

Consider Your Feedback



As a TDD practitioner, part of your job is to gain feedback from short-cycled experiments (test cycles) and adjust accordingly. Similarly, consider it your job to continuously seek improvement: Treat each of the above variants from your normal practice as a possible experiment. Run the experiment fairly, and see if the variant adds value to your TDD repertoire. If you hate it after a fair shake, drop it — that's fine, too!

Test First: TDD, BDD, and Unit Testing

Five Myths About Test-Driven Development

Erik Dietrich, DaedTech LLC

I've seen a lot of misconceptions, or myths about test driven development over the years.

Why? Well, it's hard to say directly why. Indirectly, it's easy. I've spent a lot of the last half decade not just practicing TDD, but teaching it. This includes video courses, blog posts, and working with clients specifically to teach their teams the practice. I have a lot of exposure to people about to learn the practice, so if anyone were going to hear the myths about it, I would.

But it's harder to pin down where they come from and why. My personal hypothesis is that TDD has emerged, over the years, as The Right Thing ™. So anyone not doing it these days has a strong incentive to do one of two things.

- Come up with a valid reason not to do it (i.e. "why TDD is bad").
- Hand-wave a bit to manufacture experience on the subject.

Let me be clear about something. This is **entirely rational behavior** in a situation where one feels some pressure and perhaps even existential doubt about their job. But it also serves to muddy the waters. So today, let's remove some of the mud and clear things up. Let's look at some myths about test driven development.

Myth 1: TDD is a Synonym for Unit Testing

"Does anyone have prior experience with TDD," I often ask.

"Oh, yes, I'm familiar with it. We wrote 'JUnits' and asserts at my last position, because the build required minimum test coverage."

I've lost track of how many times I've had this very exchange, modulo the particular test framework, language, and code coverage tool. The message is the same, though. "Yes, I wrote some unit tests once."

Writing unit tests does not mean that you've practiced TDD anymore than owning a wrench means you've changed your car's oil. TDD isn't about writing unit tests. It's an **approach to writing production code** that happens to produce unit tests. And it has a very specific cycle of steps that you follow.

Describing those steps in depth would take us beyond the scope of this post, but suffice it to say that it involves writing unit tests, writing production code and refactoring production code in a very orchestrated and specific sequence. Slapping a few unit tests into your codebase right before committing doesn't qualify.

Myth 2: With TDD, You Write All Tests Before the Production Code

Remember how I just said that there's a specific sequence of activities for test driven development? Well, guess what that sequence doesn't ask you to do. It doesn't ask you to write every test you might conceivably need and then start on your production code, in some kind of waterfall-esque approach within the implementation phase.

People frequently object to the idea of TDD on the grounds that writing all of your tests firsts is silly and wasteful. And they're completely right. That would be silly and wasteful. Luckily, TDD doesn't ask you to do that.

Test driven development involves writing a single test that fails, and then adding the code necessary to make that test pass. You add tests to your codebase just as you add production code — incrementally and practically.

Myth 3: TDD Practitioners Don't Do Design or Architecture

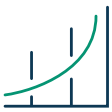
This particular myth has a bit of contrast with the ones I've mentioned so far. Those originate purely from people that haven't ever actually tried TDD. This one, on the other hand, gets some help from the occasional person that has.

This often happens with people at a company undergoing an agile transformation. They take the principles of YAGNI, emergent design, and TDD, kind of ball them all together, and conflate them with their previous world of a gigantic, seemingly endless "design phase."

"Awesome! TDD means we don't have to do that anymore! We can finally just start writing code and not worry about anything else." Detractors of the practice then seize on this sentiment as some kind of failing inherent to TDD.

It's not. That's a myth. TDD is, again, a sequence by which you write code when you're ready to start writing code. It makes absolutely no prohibition against thinking through your eventual design, looking out for pitfalls, or whiteboarding architecture. You should absolutely do these things, TDD or no TDD.

Myth 4: You Should Do TDD to Increase Your Test Coverage



Frankly, I could write an entire post about the subject of test coverage and why this is a metric that nobody outside of the dev team should look at. But this isn't that post, so I'll just briefly state that I find test coverage to be a frequently problematic metric and most definitely not a first class goal. Your team's test coverage should serve only to tip the team off as to where it has untested code.

In light of that take, you can understand why I cringe at the idea that test coverage is a first class goal and that TDD's purpose is to improve it. No, no, no!

TDD provides an awful lot of wins: a robust regression test suite, the ability to fearlessly refactor your code, the avoidance of writing unnecessary code, and plenty more. Those are first class benefits. Test coverage is just a trailing indicator that this has happened, if anything.

Myth 5: TDD is a QA Strategy and Can Even Replace QA

I'll close with perhaps the most business-focused myth that I hear, and potentially the most damaging one. If you're trying to adopt TDD as an attempt to move QA into the development team or to cut cost by reducing the QA department, you're headed down a dangerous path.

The term is "test driven development." It's a software development technique — not a QA approach. Perhaps the most philosophical and fundamental way that I can describe TDD is to say that it involves breaking your code into tiny pieces of functionality and then carefully defining "done" before you start the next incremental piece. Write a test that fails, but you know that, when it passes, you'll be done with the current bit you're working on.

This means that TDD produces exactly as many tests as it takes to get to done, and not one more. So no comprehensive edge case testing. No smoke tests and no load tests. No exploratory tests. You swap TDD for QA at your extreme peril.

This, like all the other myths, involve fundamental misunderstandings about the nature of TDD. TDD is a simple but powerful technique for writing code, and one that happens to produce a lot of collateral good. So do yourself a favor and give it a serious try before jumping to conclusions about what it is, what it isn't, and what it lets you do.

How Does BDD Impact Your Testing Strategy?

Erick Dietrich, DaedTech LLC

So, what is this BDD testing stuff, anyway? Before I answer that (as it turns out, nonsensical) question, I'll speak briefly about the sometimes-frantic world of software development trends.

It seems that, methodologically, software development reinvents itself at a staggering pace. First there was software development, and then there were formalized processes, like the Rational Unified Process (RUP). Then we went agile, but that wasn't quite enough, so we scaled agile, got lean, and started doing something ominously called "mobbing." And that's just on the process side, with workflows and collaboration models.

When it comes to development techniques, we like to let things drive development and design. The last couple of decades have brought the emergence of test **driven** development (TDD), acceptance test **driven** development (ATDD), domain driven **design** (DDD), and behavior **driven** development (BDD). This all makes for a fairly manic pace of change.

I attribute this largely to the relative youth of software development as a profession. Things like accounting and physics have been around for hundreds of years, so the basics have solidified some. With software, we're still working our way there.

And this frenetic pace is a double-edged sword. It's good because we're rapidly evolving, improving, and maturing. But it's a challenge when you need to separate the important developments from the flashes in the pan. And it's especially challenging for those in collaboration with the software developers, trying to understand how changes to software development techniques impact them.

And so we arrive at the central question of this article:

If the development organization starts to make noise about BDD,
how does it impact you? How does it impact the overall testing strategy?

To Understand BDD, Understand Unit Testing

Don't worry. I'll get to what BDD is shortly. Before I can do that, though, I need to cover an essential prerequisite: unit testing.

If you earn a living testing software, there's a pretty good chance that you've heard the term "unit testing" in regards to something that developers do. There's also a good chance you've regarded it somewhat suspiciously, wondering if the developers aren't wasting time doing your testing job instead of, you know, developing the software.

But, as it turns out, there's no conflict here. Developers are, in fact, doing a form of testing when they do this. But they're doing something both very necessary and very granular, and it involves writing code.

A quick analogy will help understanding. As hypothetical "car tester," you might check on things like the following:

- Does the car start when I press the ignition button?
- At highway speeds, does the car run normally or is it noisy?
- Do all of the doors and windows open and shut easily?

You get the idea. If this is your job as a "car tester," then here is what the "car developers"

are doing when they write unit tests.

- Does this particular engine component heat to this particular temperature?
- If we apply a certain amount of current to the dashboard light fuse, does it safely blow out?
- Are these screws Phillips head?

Unit testing is the developers checking their work with automated tests. They write these tests and run them, and they're so granular, specific and low-level that they make sense to no one outside of the software development group.

What is BDD?

With a definition of unit testing in the books, we can now make our way toward BDD. Unit tests are granular, automated things that run quickly and test code in isolation. And, a couple of decades ago, some pioneers of the TDD technique had the idea to write these tests as they wrote code instead of afterward.

Test driven development (TDD) has many benefits, all of which are beyond the scope of this article. So without delving into the motivations, let's just say that TDD forces you to articulate with a test what the code should do, before you actually write that code. In a sense, it's like the scientific method. Before you start with the "experiment" of writing your production code, you form a "hypothesis" of what the result should look like.

Behavior-driven development (BDD) is an evolution of TDD. It's a development approach that produces more business-centric tests.

TDD practitioners enjoyed the benefits and cadence of the practice, but started to think, "what if we applied this beyond the most granular, nuts and bolts concerns? What if we articulated requirements in natural language and followed the TDD approach? And what if we involved other stakeholders outside of the development group?"

A BDD scenario might proceed this way.

1. You have a user story that someone should be able to log into your site.
2. You take that story and express it in "given-when-then" format. "Given a valid user account, when I submit its username and password, then I should see the logged in homepage."
3. The development team writes code that translates this statement into an actual automated test that can pass or fail, and it starts off failing.
4. They write code until it passes.

How BDD Affects Your Testing Strategy

So with all of that in mind, how does BDD affect your testing strategy? Well, in the very beginning of the post, recall that I said that "BDD testing" was nonsense? It's time now for an explanation.

Behavior driven development is a development technique that happens to produce automated tests as a byproduct. So "BDD testing" is nonsense because BDD is not truly a testing strategy, but rather a way to define and verify that a requirement is complete.

And that tells you everything you need to know about the impact on your testing strategy. Since BDD is not a software testing strategy, per se, you still need your testing strategy. You still need exploratory testing and regression testing. You still need smoke testing, load testing, and performance testing. And you still need intelligent humans to do all of these things.

BDD doesn't give the QA folks an extra thing to do or an extra task to complete. Instead, it involves them earlier in the process and gives them an important seat at the table during discussions of "how should this behave and how do we know when it's done?" BDD fits into your testing strategy by forcing agreement from all stakeholders on what, exactly, to test.

Why BDD Helps

Let's take a breath and think about this for a moment. It has some weighty implications.

When you successfully execute this approach, you have a natural language expression of a requirement and an automated test that passes or fails depending on whether or not the requirement is satisfied. This means that you have a very specific definition of done for each requirement.

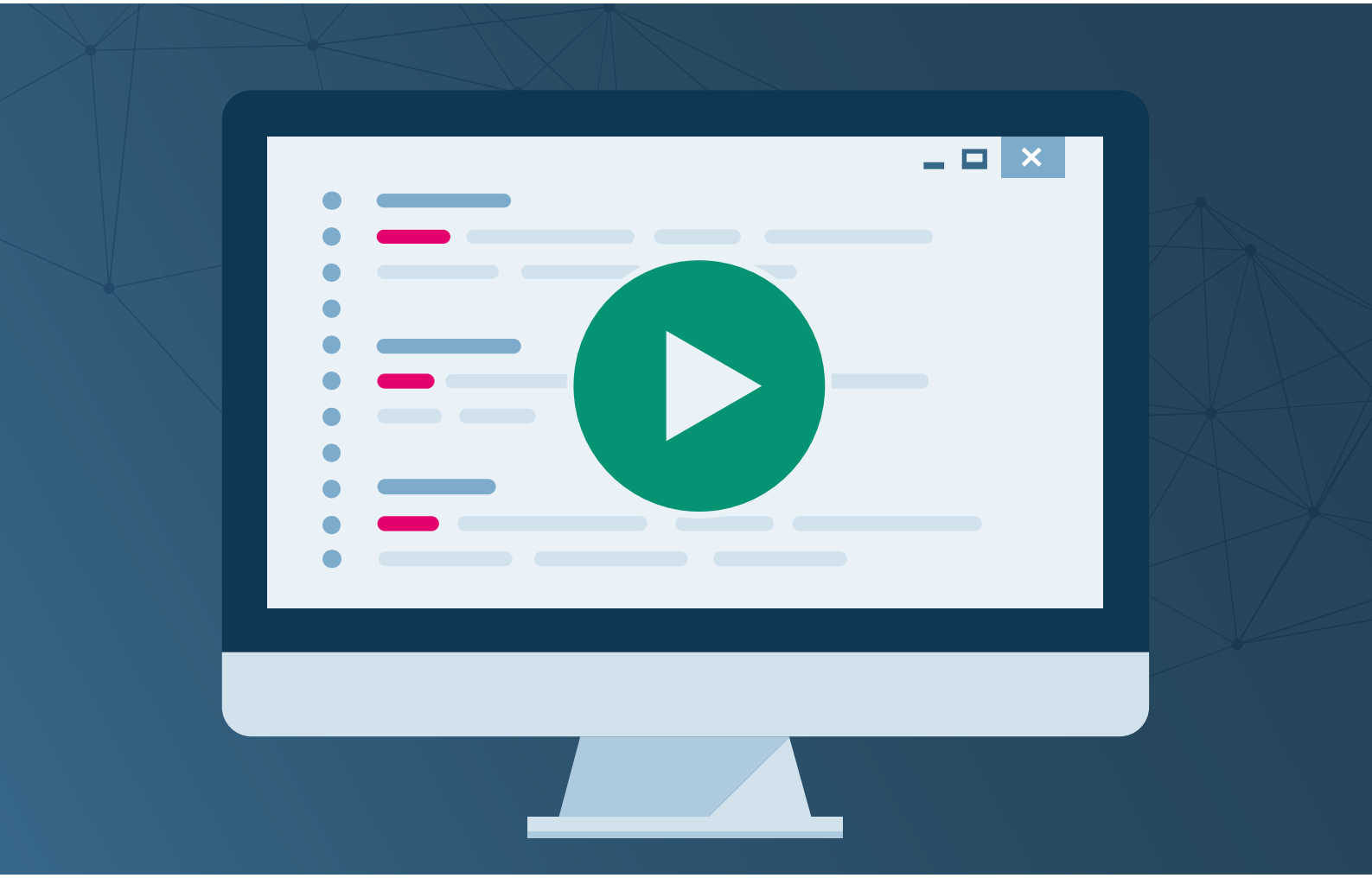
How many meetings (arguments) have you had over the years where different stakeholders in the software development process argue over whether the software satisfies a requirement or not? I bet it's more than you can count.

"Look, it does what the spec says. Users need to be able to log in, and they can log in. There's no bug."

"What do you mean there's no bug!? It takes them back to the login page! They have to manually type in the home page URL, and they have no way of knowing they're logged in!"

"Well, technically, that does fit with the spec."

When you follow a BDD approach, you bring the stakeholders together ahead of time and get them to agree on what it means to satisfy a requirement; exactly what it means. The developers then codify this into an executable and measurable test. The corpus of all of these BDD tests then tells you at a glance whether the software satisfies all requirements or not.



Test First: TDD, BDD, and Unit Testing

BDD in Action

Justin Rohrman, Excelon Development

Behavior-driven development (BDD) is part of the development process in my current gig. I had lightly used BDD frameworks years before, but never the process, and we hadn't heavily integrated it into our development cycles. I spent some time researching, hunting for authentic stories about using the process and tools, but I only found definitions and information about programming libraries. I couldn't find an in-depth experience report.

I want to start writing that now. This is what I have found:

The Conversation

BDD provides a simple framework for describing state, transitions, and outcomes through a given-when-then convention. These three words are the focus of BDD. "Given-when-then" is a tool to frame and constrain how conversations in software projects happen. Yes, constrain — my experience has been counter to what most BDD leaders promote.

My experience with the conversation aspect of BDD is very similar to my experience with Scrum. These frameworks are where you begin if you are at the beginning of developing a team. If you are starting from nothing, your technical team doesn't know how to talk about product usage, so BDD is a place to start. But once technical teams are accustomed to talking about customer value, how the product is used, and how we might test to discover problems, then it is probably time to break the convention and discover what works for the team.

If having conversations is more important than anything else that happens in the context of BDD, I want to have conversations.

We initially started using BDD by talking through the given-when-then framework. This convention allowed us to talk about the specific aspects of a change that a customer values:

- **Given** I am on a new form
- **When** I populate a signature field
- **Then** all fields on the form become disabled

This small bit of text helps us think about what a person wants to do with our software, but it also sets off my tester alarms. Testing, to me, is an open-ended activity. Questions immediately pop into my head when I see a new page, or even a new field on a page. In BDD terms, we might say:

- **Given** I am on my user profile
- **When** I update my birthdate
- **Then** my birthdate is displayed under my avatar

Without the BDD framework, I automatically start thinking, what is a date? What is a good date? What happens when I enter a bad date? Can I make the date display under the avatar "misbehave" somehow? I have never seen BDD as a development practice capture these questions.

What I do see is anchoring on simple tests or demonstration scenarios. We start working on a change with a stubbed BDD scenario, get to the point where that scenario passes, and then we may think we are done. If we're not careful, BDD can make it easy to forget that there are a lot of unanswered questions we should be asking.

The Tests

While I often hear stories about tests being built as a questioning framework, we build ours as bits of automation during a development cycle. One of the strong points of my current team is that the question "How do we test this?" is often one of the first things we ask. This leads to more careful design and more testable (and usable) software. So, when we start a new change with an important UI component, we will often start by building a BDD test.

This begins by talking through the change: what we are doing, what the customer is expecting to get out of this, what should be done at the unit level, and what is best left to test in the browser. Most often, we settle on one or two scenarios to build against the browser using code.

Let's use my birthdate example once more. I would start by thinking about the state I need the software to be in to test this, and also whether there is code I can reuse — or, rather, whether there is a step, and maybe a bit of page object, that already exists that we can use so we don't have to write new code. At this point in the life of our test suite, the answer is often yes.

I take the step that defines my initial state, the "given" part of the scenario, and move from there. The next questions are "What actions do I want to perform?" and "What assertions do I want to make?" Because we have a new date field, I probably have to update my page object to define that field and make it available for tests. If I want to develop my test with customer value in mind, I'll start from the outside in by creating the plain-text step first: when I set a birth-date.

After specifying what I want to do, I'd open my step file and create a new step that sets a value in the date field. At this point you have a choice to make: do things the easy way and set a value in the DOM, or do things the realistic way and use the date picker. I want realism, so I write a method to open the datepicker and set a value.

Once we can manipulate data, we want to make an assertion that the data persists. In our case, we want to do that in two places: once in the date field, to make sure that our selected date displays, and once where the user avatar displays.

If I had a chance to rebuild this framework from the ground up, I would completely dispense with Cucumber. The notion that nontechnical members of the development team, such as product owners, will write and review tests using Cucumber plain-text syntax is approaching absurd. For me, Cucumber adds an extra layer or two of abstraction, namely the feature files and the step files. Doing away with those would make a more maintainable test framework that most slightly technical people would be able to use and understand.

I will say, though, that having the steps in plain text makes tracking failures in continuous integration easier. Rather than seeing a stack trace and a questionable line number, I can see that a test failed on "When I set a birth-date."

The Value

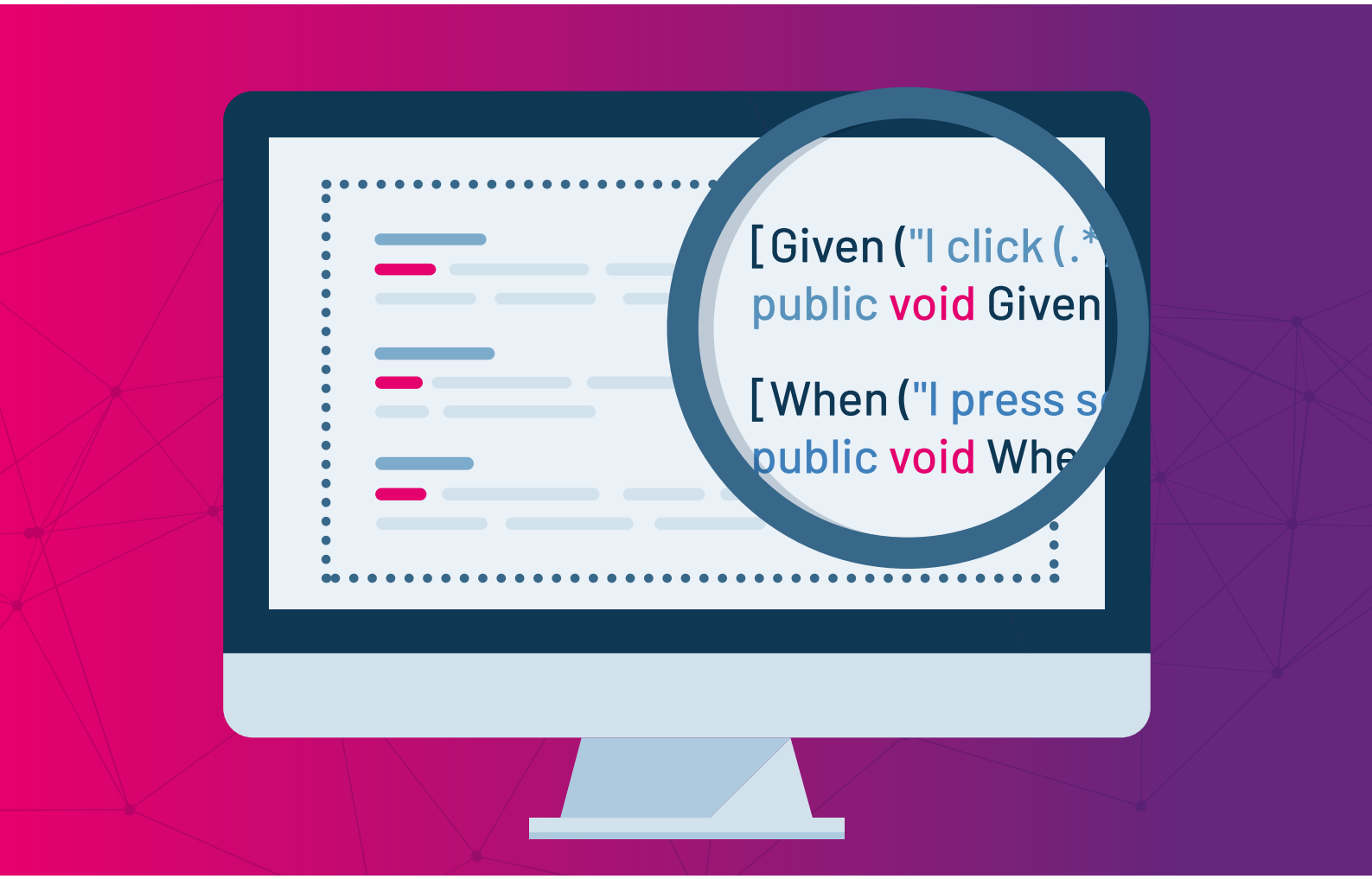
BDD practitioners regularly tell me that the most important part of these tests is the conversation, but at this point, I'm not sure I agree.

I work in an environment where we make small changes and deliveries very frequently. We work in pairs and can deliver small changes to production every two to three days, on average. We are also a refactoring machine. Improving the code, whether test code or production code, is part of the ethos. That translates to a lot of code, as well as new risk being introduced frequently, however small a change may be. Once we make a change and have passing tests at appropriate levels in the tech stack — unit, service and browser — we take a look at what can be refactored. That usually means breaking something that worked before. Having those layers of tests, including the UI, will often tell us we have gone astray before we make a build and can perform exploration.

I see BDD tests fail often — maybe not daily, but often enough for the value to be completely obvious. We like to pretend that we can understand every state a piece of software can get into — fields will only ever have these values, the customer will only ever follow these workflows — and we are regularly surprised. It is hard to predict how a change will affect other parts of our product. Tests built with BDD help us discover these problems quickly.

Using BDD

My experience in BDD is different from many of the stories I hear, as are the things I value about it. But I enjoy it and appreciate it as part of my testing practice. BDD helps us keep customer value in mind, drives simple code design, and provides a fast feedback loop for refactoring and future changes.



Clarifying Scope with Scenarios in BDD

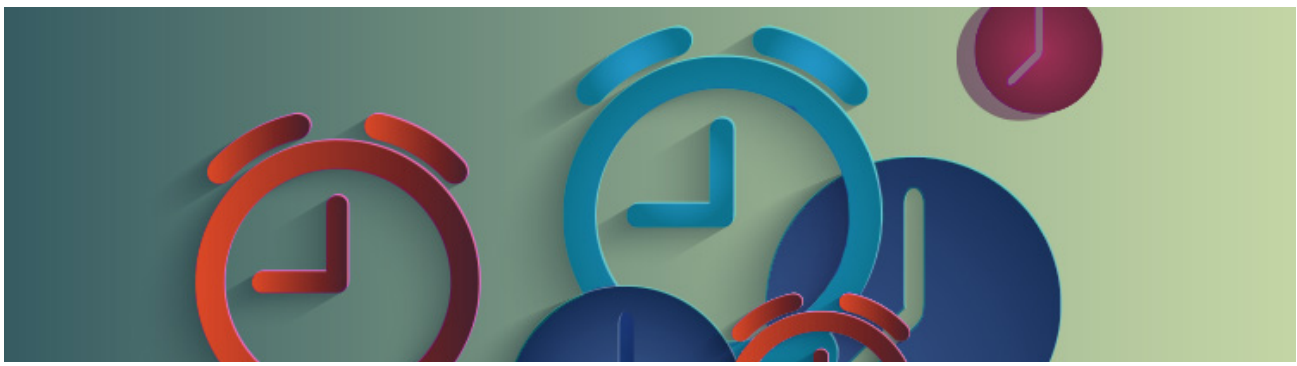
Jeff Langr, Langr Software Solutions

It doesn't take long for teams to learn just enough about behavior-driven development (BDD) to be excited. I used to sit for a couple hours with business analysis, developers, product owners, testers, and even managers, showing them the fundamentals of BDD. My goal was to help them hit the ground running by answering questions for them: What are we trying to do with BDD? What does a feature look like? What's a scenario? What is Gherkin Language and how do I write my tests? Where's the value in BDD for me, for my team, and for my organization?

We would work through a few BDD examples together, then they'd disappear to a meeting. The next time I checked up with them, perhaps the next day, they invariably had slammed out a good number of scenarios. Great! We would sit and talk for a while about how to clean things up a bit. They would then disappear and return with even more the next time.

As long as I stayed on top of my "would-be" behavior-drivers, this back-and-forth seemed productive and straightforward. If I touched based with them soon enough, it was easy to correct the various, common problems that would arise. However, I eventually realized that I was leading my learners down a not-very-agile path.

Embracing a "Just in Time" Mentality



One of the aspects I learned to appreciate about agile is the just-in-time mentality it promotes. We defer as much as we can, until as late as possible. Waiting until the last responsible moment sometimes results in the joyful revelation that the effort is no longer needed, or that we avoided having to rework something already solved. The gift of time, earned by the simple acts of demonstrating patience and avoiding speculation.

Asking folks to go off and produce piles of given-when-then's isn't very "just in time." Investing discussion time and angst to derive given-when-then narratives is a waste of time if a scenario gets discarded. Even if they do ultimately build to a scenario, the details are reasonably likely to change between now and then.

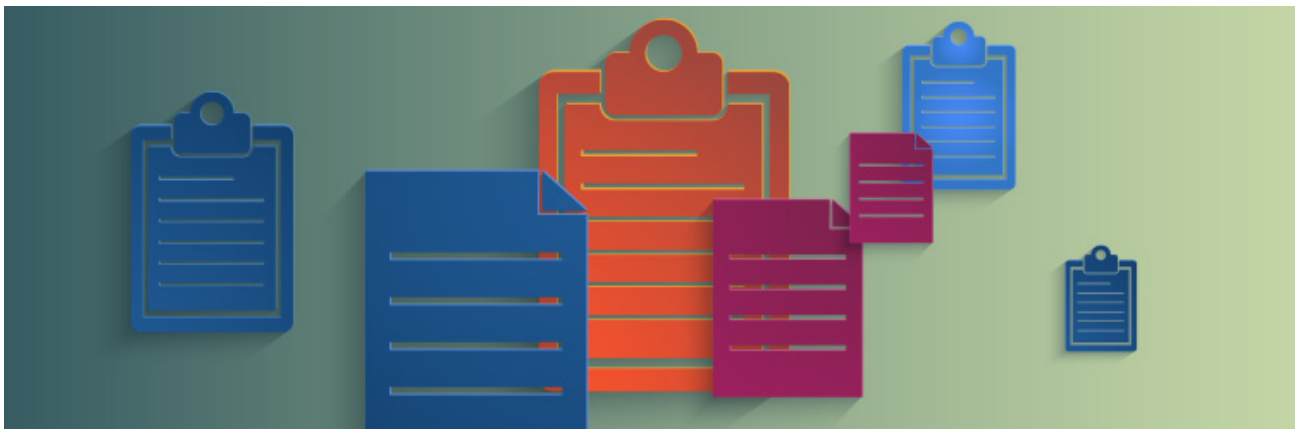
Negotiating the Behaviors

Part of our goal in producing given-when-then narratives (I'll call these the "specs" moving forward) is to ensure we, the organization asking for capabilities and the team delivering them, are all on the same page. Try viewing deriving the specs as an agile form of contract negotiation: If the business agrees that the specs represent their interests, and if we deliver a system that meets all those specs (i.e. that passes all the tests), the business agrees to buy it.

When do the specs become a binding contract? The simple answer: When we agree to take on, and deliver the behavior they describe. If we're employing an iterative agile process like Scrum, that moment is at the outset of the iteration when we take on the work. Even then, testers and developers can continue to negotiate right up until the moment we deliver the goods. It's software, so nothing needs to be finalized until it's shipped... and even then, we can agree to change it. In agile, we negotiate our "contracts" continually.

Until the moment we're considering taking on the story, we do not need to flesh out all narratives for its scenarios.

Do We Need A Narrative to Discuss Things?



Agile is an incremental, iterative process. The "iterative" part means that we continually refine things: we start at high levels, and then cycle down toward the low-level details needed to ship a product. With respect to behaviors, the business desires (or "requirements"), we start with a story. Never mind folks who think "story" is a synonym for "requirement." Think instead of a story as the real-life thing: it's a discussion that begins with the business telling us about what they'd like.

A story starts as a simple tale, but gets more interesting and detailed as we talk. We flesh out our understanding of a story by asking questions: "What happens in this case?" Our questions are often answered with summary descriptions:

> *"We're working on the library feature that allows people to self-scan and check out movies. The system prevents underage patrons from checking out adult movies. However, half of the kids are sneaking off with the movie. What should happen?"*

> *"Oh dear! I guess we need to worry about that. It should probably send a notification to the librarian's machine at the desk."*

> *"OK. We've noted a scenario titled: movie checkout by underage patron sends notification to librarian."*

We collect the summary descriptions; these become our scenario titles. When we're closer to building, we can do iterative refinement by providing specific given-when-then narratives for each scenario.

In the meantime, however, the scenario titles might be all we need. It takes but a smidgen of imagination to think about what the narrative might be for many scenarios, including the one above where a child checks out a restricted movie.

When we discuss the desired behaviour, maybe just prior to the outset of an iteration, we want to get some sense of what this thing really is, and accordingly, how big it really is. Deriving solely the list of scenario titles can quickly help us to agree the scope that we're ready to tackle now.

Suppose for the checkout feature we have a handful of basic scenarios… then someone remembers to ask the question, "But what about the underage patrons?" Oh. We now we have a different picture in our mind. We quickly realize there are at least a handful more things we must consider: we must add ratings to the movies, we must ensure we capture the patron's birthdate, and we must ensure that our new notification feature doesn't impact adult patrons or underage patrons checking out non-adult materials.

Do we need to detail these new scenarios? Not yet! We realize that the story no longer represents something small. The product owner decides that we can worry about those needs in a later iteration. She has something more important for us instead. Effort expended on detailing the specs for the underage patrons: none.

Occasionally we do need the detailed narrative in order to gain better understanding ("What do you really mean? Show me an example.") or make planning decisions ("That could work one of two ways… let's talk through the two variant narratives"). Spending the time to provide details for an occasional scenario is fine. What we want to avoid is always diving deep before we've explored the breadth of a story. And from a timing perspective, the deep-dive into narratives can occur immediately following the use of the scenario names to determine scope.

Next time you start a conversation about a feature, first pin down the list of scenario names, and use those as a basis for the discussion.

About TestRail

We build popular software testing tools for QA and development teams. Many of the world's best teams and thousands of testers and developers use our products to build rock-solid software every day. We are proud that TestRail – our web-based test management tool – has become one of the leading tools to help software teams improve their testing efforts.

Gurock Software was founded in 2004 and we now have offices in Frankfurt (our HQ), Dublin, Austin & Houston. Our world-wide distributed team focuses on building and supporting powerful tools with beautiful interfaces to help software teams around the world ship reliable software.

Gurock part of the Idera, Inc. family of testing tools, which includes Ranorex, Kiuwan, Travis CI. Idera, Inc. is the parent company of global B2B software productivity brands whose solutions enable technical users to do more with less, faster. Idera, Inc. brands span three divisions – Database Tools, Developer Tools, and Test Management Tools – with products that are evangelized by millions of community members and more than 50,000 customers worldwide, including some of the world's largest healthcare, financial services, retail, and technology companies.