The Ultimate Guide to Testing and BDD

TestRail

The Ultimate Guide to Testing and BDD

Contents

Sign up for our weekly Testers Email here on our blog!

SIGN UP HERE!

How Does BDD Impact Your Overall Testing Strategy?

This is a guest post by Erik Dietrich, founder of DaedTech LLC, programmer, architect, IT management consultant, author, and technologist.

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 post.

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 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 post. 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.

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 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 these BDD tests then tells you at a glance whether the software satisfies all requirements or not.

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 by-product. 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 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.

Clarifying Scope with Scenarios in Behavior-Driven Development

Article written by Jeff Langr, founder of Langr Software Solutions, Inc.

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.

If 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 and 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.

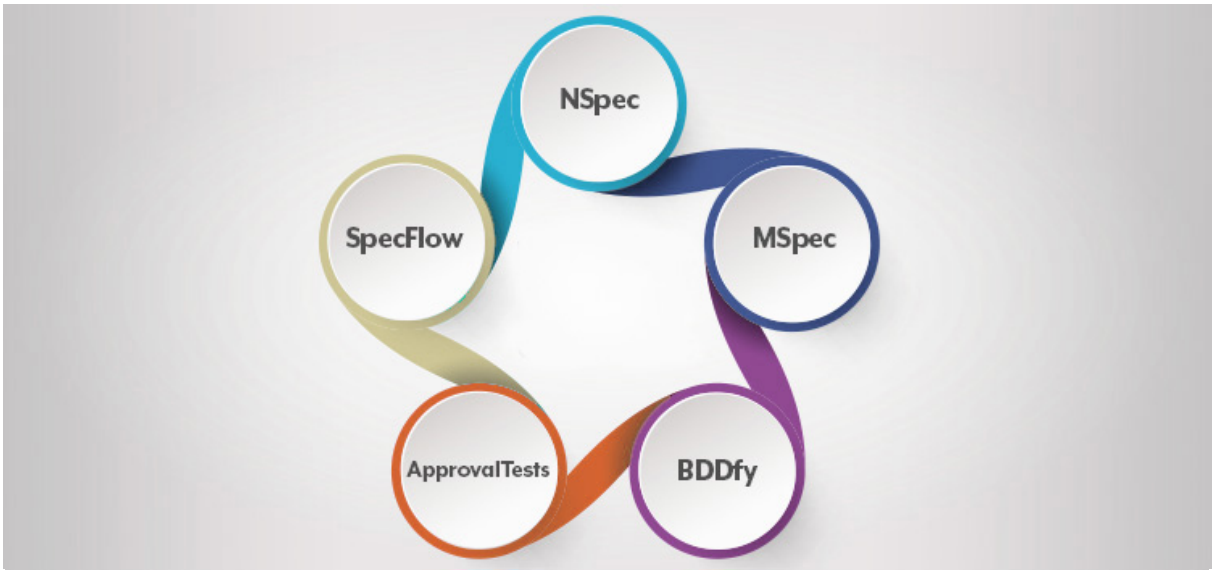
Do we need to detail these new scenarios? Not yet! We realize that the story no longer represents something small. The product owner decides that we can worry about those needs in a later iteration. She has something more important for us instead. Effort expended on detailing the specs for the underage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

Occasionally we do need the detailed narrative 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.

A Look at 5 BDD Tools for C# Codebases



Written by Erik Dietrich, founder of DaedTech LLC, programmer, architect, IT management consultant, author, and technologist.

You could be forgiven for rolling your eyes, initially, at the idea of BDD. In the first place, with TDD, DDD, ATDD, and BDD, the software world seems really to have loaded up on things that end in DD. Add to that some lofty promises about bridging the gap between code and domain language, and you have a recipe for scepticism.

But resist this impulse. BDD is hugely helpful, given the right tools and the right process.

The overarching aim is simple but powerful. First you express what you want the system to do in natural language, so that everyone understands. Then you match that behavior to automated tests.

It sounds simple, but the devil is in the details. Let's explore some tools to help you achieve this, while making those details less devilish. Specifically, let's look at some BDD tools for C#.

NSpec

First up, let's examine NSpec. NSpec comes from the so-called "spec" flavor of BDD.

For a detailed treatment of "spec" BDD vs "behave" BDD, you can check out this Stack Overflow question. But briefly, behave flavored tools focus on creating actual

artefacts with English language in them, and then mapping that language to code. The spec style tools have the same conversational focus, but they emphasize creating human-readable names for elements of code, such as classes and methods. And on that spectrum, NSpec is, not surprisingly given its name, a spec flavored tool.

Install is easy enough. You can get started simply by installing the NSpec NuGet package, and you can integrate with your existing unit test runner. NSpec works by building up incremental context.

That's a fancy way to say that you can divide up the various setup routines that are relevant for your testing; and assign nice human readable names to them. Most unit tests work completely context-free, in isolation, so this approach lets you rapidly construct and reuse scenarios.

MSpec

No, that's not a typo or a repeat. This second of the BDD tools does, in fact, differ from the first, but only by a letter. MSpec is an open source, spec-flavored BDD framework.

MSpec stands for "machine specifications," and differs by only a letter from NSpec, which draws from the curious .NET ecosystem trend to name everything N-something. Like NSpec, MSpec also uses source code semantics to allow incremental context specification. As its charter, it seeks to eliminate as much of the noise and boilerplate as possible around automated testing and leave you with just the bare and readable essentials. This framework relies on convention over configuration.

As with NSpec, you can install MSpec simply by using NuGet (it will appear as Machine.Specifications). You may also want to pair it with an assertion framework. MSpec has its own, and it can also work with popular fluent assertion library Shouldly.

BDDfy

Having looked at two spec flavored BDD tools, let's mix it up with a behave style tool. (Though, this one is both). I'm talking about BDDfy by Test Stack. BDDfy, as you might infer from its name, is a BDD framework and one that bills itself as being dead simple both to use and to extend.

BBDfy is interesting in that it can run with any unit testing framework, but you can use it without even using such a framework. It's truly flexible. And this extends to its philosophy with the flavors of BDD as well. You can employ its spec-flavored, context building functionality. But you can also then overlay it with so-called "stories" where

you map the concept of agile user stories to units of code. This is how it becomes a behavior flavored BBD tool.

You'll probably notice a theme here in that you install BDDfy with NuGet as well. You can also install a NuGet package that gives you usage examples as well.

ApprovalTests

Next up, let's examine ApprovalTests (also on Github here). ApprovalTests falls more into the spec end of the spectrum as far as BDD tools flavors, but it has some novel elements to it.

The idea here is to model and test more involved bits of functionality than unit tests allow. In that sense, you have the same concept of NSpec and MSpec for constructing named scenarios with context. But with ApprovalTests, you introduce the concept of approvals. The name isn't a non-sequitur by any stretch.

Here's the idea: In code, you map your ideas in natural language and then turn those into test code, which you use to generate production code (in essence, making use of TDD). Then, your test suite captures the result of that production code in a file that describes the result in detail. And, finally, you approve the result when satisfied, which stores it for later. Changing an approved test means changing your requirements.

As you can probably guess, you can find ApprovalTests on NuGet, and it works with common test frameworks.

SpecFlow

The last framework I'll cover is the de facto incumbent in the .NET world. If you search around and ask around for BDD tools in C#, you will hear about SpecFlow. SpecFlow falls firmly on the behave end of the spectrum of BDD tools.

Here's how SpecFlow works. It follows given-when-then (GWT) syntax to map English statements to source code methods. The GWT statement reside in so-called feature files that are simply text files. The SpecFlow framework then uses source code configuration constructs to determine which code the statements should trigger. The end result… You have an automated system for running tests that non-technical people can easily read and discuss.

As with everything else, you can find SpecFlow on NuGet. But keep in mind that you'll also want a Visual Studio plugin to help you run your SpecFlow scenarios.

Go Out and Experiment

So, which of these should you use? It's hard to say. They're all good tools, so I'd say go out try each of them for a bit and see which one appeals to you. None of them has a particularly steep learning curve, particularly if you're already familiar with automated testing.

In the end, it's less important which specific tool you use and more important that you start having meaningful testing processes. BDD is a powerful tool for getting the business, testers and developers all in agreement on what it means for software to be done.

Should We Automate All Our Feature Scenarios?

Article written by Jeff Langr, founder of Langr Software Solutions, Inc.

Behavior-driven development (BDD) is a methodology in which we specify an application by first describing its behavior in the form of human-readable examples. Here's a simple example that describes how one element of ATM withdrawals should work:

- Features: ATM actions
- Background: The account holder has successfully selected an account using their debit card
- Scenario: Rejects attempt to withdraw more funds than available in the account holder's account:
    - Given the account holder has an account balance of $100,
    - When the account holder attempts to withdraw $110,
    - Then the ATM displays the message "Insufficient funds; transaction cancelled",
    - And the ATM ejects the card.

If you've built software long enough, you might note some similarities to the concept of using use cases to describe the requirements of a system. So, what's different? On the surface, not much. The terminology (features and scenarios) is a little different,

but a *scenario* in BDD is much like the *narrative* of a use case: An actor (in the ATM example, the account holder) interacts with the system, and the system responds in some accordant fashion. The use case goal (its title) is similar to the scenario summary description- both represent the end state that should occur when an actor interacts with the system.

Use cases were intended to be a different way of driving the design and implementation of a system from its requirements. The set of use cases represents the comprehensive set of end goals that all possible actors can accomplish with a system. The narrative form provides the best way to capture and describe the steps (requirements) needed to accomplish these end goals. Wikipedia tells us that the "main success scenario of each use case provides everyone involved with an agreement as to what the system will basically do and what it will not do."

The key word in the Wikipedia description of use cases is agreement. In order to deliver successful software, we- the product owners, testers and developers, must all be on the same page regarding how it should behave. Use cases give us a focal point for agreement. We can quibble about the narratives and pin down a new one: "No, the ATM should not eject the card; it should instead prompt the user for another action."

Still, there are a couple of key differences between use cases and the examples of BDD:

- Organizationally, BDD scenarios don't match up one to one with use cases; a discrete scenario might be an alternate case for a single use case.

- Use cases aren't specific examples of one user's interactions; they attempt to generalize the behavior. For our ATM example, a use case might look like this:

    - Actor: Account holder
    - Goal: Withdraw money from account
    - Narrative:
        - (Happy path withdrawal case here)
    - Alternate path:
        - The account holder attempts to withdraw more funds than available in the account
        - The system responds with an error message and ejects the card

Note that the alternate path narrative for this use case doesn't mention specifics (e.g., an account balance of $100).

Where Did the Use Cases Go?

I rarely encounter teams incorporating use cases into their process anymore, so something must not be so great about them. What's not to like?

Well, for one, teams often got into what might be called "analysis paralysis." They debated the use cases endlessly, sometimes in an obsessive-compulsive manner. I remember at least a few debates over the granularity of the use cases, and worse, sometimes we would argue incessantly over wording: "We should use the word will instead of should here."

The bigger problem is that the use cases are often misinterpreted. No matter how much we wordsmith, it's possible to misunderstand the English language. Take this classic programmer joke:

> *A programmer is going to the grocery store. Her spouse says, "Buy a gallon of milk, and if there are eggs, buy a dozen." On arriving back home after the grocery trip, the spouse angrily asks, "Why did you get 12 gallons of milk?" The programmer replies, "They had eggs."*

One customer described to me their painful experience with use cases. The product owner had sent some use cases to an offshore team for development; what came back a couple of weeks later wasn't quite what they expected. Through a series of updates to the use cases, the development team kept trying to build what they thought the product owner was asking for. Unfortunately, this back-and-forth took three more two-week iterations before the developers got it right.

Resorting to "legalese" provides one way to remove ambiguity, but we then often produce inscrutable prose that no one can understand. Using examples in BDD can begin to minimize opportunities for misinterpretation. Here's a tableized set of examples that represent the intended outcome for the grocery trip:

Has eggs?	Purchase
No	1 gallon milk, no eggs
Yes	1 gallon milk, 12 eggs

Granted, the example is a little silly, but distilling the ambiguous prose to a set of expected outcomes for several scenarios often provides immediate clarity. As a member of the development team, if you're still uncertain about the appropriate behavior, you can always request more scenarios from the product owner.

One Step Beyond

Most teams adopting BDD use a tool such as Cucumber or FitNesse to support automating these examples. The ATM example from earlier is written to support use of Cucumber. As such, each step of its scenario is automated to support an executable test.

Behind the scenes, each step description (for example, “When the account holder attempts to withdraw $110”) gets translated into code that drives the system you’re testing. Steps that verify things (indicated by statements starting with “Then,” such as “Then the ATM displays the message”) get translated into code that asks the system a question, then compares the answer with the expected outcome. If the answer matches the expectation, the scenario passes; otherwise, it fails.

Automating BDD scenarios allows a development team to demonstrate that they’ve met product owner expectations. We can execute these examples at the click of a button. If the scenario examples all pass, the development team knows their work is complete; if any one fails, the development team knows they have more work to do. The rapid feedback provided by running these automated examples supports our interest in short-cycle iterative development: After adding a new feature, we can gain the confidence to ship it by making sure all the examples execute successfully.

Oh, No- We Can’t Automate This!

Automating the examples makes continuous delivery feasible. But the most important value we get from crafting the scenarios in BDD is their ability to act as a focal point for understanding and negotiation. This conforms to our Wikipedia description for use cases as “an agreement as to what the system will basically do and what it will not do.”

Sometimes it might not even be possible to automate any scenarios. Many years ago, I worked to help a customer introduce BDD into their process for a production web-based system that they were continually updating. I sat and worked through a few feature examples with the business analysts to help them understand what they should be doing. The customer went off on their own and proceeded to flesh out numerous additional scenarios. Meanwhile, I attempted to automate the first scenario.

Unfortunately, my customer’s system wasn’t architected with testing in mind. I made several attempts to find a clean entry point that would allow us to automate the examples. But there were no clean APIs to hook into, and the web front end had some complexities that made it impossible to easily “robot-drive” the system through its user interface.

I felt awful about having to tell the BAs that we couldn't automate anything. We'd already spent many hours introducing BDD, and I had to tell them that there wouldn't be any cost-effective way to automate the numerous scenarios they'd created.

I was surprised when they told me not to worry. They were still excited about the BDD techniques they'd learned. The most useful proposition for them was a way to consistently approach pinning down, negotiating and organizing the requirements. They indicated that the automation would be nice, but for the time being, they found just the collaborative part of BDD to be its most useful element.

In BDD, automation is not the primary goal. Yes, automation may be essential if you want to do continuous delivery, and yes, automation is where you see great success with BDD. But the first goal is to understand how to use BDD as a means of getting on the same page. And if you can't get that far, automation isn't going to help you much anyway.

Can You Have Too Many Tests?

Article written by Jeff Langr, founder of Langr Software Solutions, Inc.

The automated tests created as part of the behavior-driven development (BDD) process provide us with numerous benefits:

- They provide a test suite that lets us know if we have any regressions.
- They give us time to do needed exploratory testing.
- They document the capabilities of the system—if the tests all pass, we know we can trust them as documentation.

With these potential benefits, it can seem that having more automated tests is always better- but that's not necessarily the case.

When to Question the Status Quo

An organization I worked with proudly told me they'd produced at least 3,500 automated acceptance tests. These tests required several hundred hours to run: The tests drove their UI and took, on average, six minutes per test. Do the math. No, wait, don't do the math; you'll be appalled.

I asked about the size of the system. What did it have to do? Roughly how many screens were involved? Was there any hidden complexity, such as a massive number of business rules hidden beneath the surface (what I call an "iceberg system")? Their answer suggested a medium-sized system—at most, a couple hundred thousand lines of C# code. Specs-by-example tests in comparable systems I'd encountered typically

numbered around 300 or 400, taking anywhere from a few minutes to a few hours to execute.

Clever scaling across a couple dozen machines allowed my customer's many thousands of tests to complete in about 11 hours each night. They'd additionally built some measures to attempt reruns of tests that might have spuriously failed, and to do other clever monitoring. The upshot: only a few person-hours expended per morning to wade through the inevitable dozen or so failing tests. (And yes, the word "only" in the preceding sentence is intentionally sarcastic.)

We sometimes are great at emulating frogs in a pot of water slowly coming to a boil. I can imagine hearing these statements uttered at some point during the evolution of this amazingly oversized test suite:

> *"The test run is taking an hour to run. It'd be nice if we could see our builds complete a few more times per day so that we could correct any problems sooner."*
>
> *"It's taking several hours for the test run to complete. We only get feedback on things we push up in the morning."*
>
> *"It's taking most of the night to run! It's barely finishing by the time we get in. And sometimes it fails completely early on, and we get zero feedback!"*
>
> *"It doesn't even finish overnight. We need to scale to two machines."*
>
> *"We need to find a server farm and pay for an additional sysadmin." (This was just a little before the rise of AWS.)*
>
> *At some point, it would have been nice if someone had dared to say, "Maybe we're doing something wrong."*

The next time things seem suspect, stop and say, "Maybe we're doing something wrong." Not much is black-and-white wrong in software development, but it's certainly wrong to let a frog slowly boil to death.

Driving the UI: A Slow Highway to Hell

A brief look at most versions of the testing pyramid suggests that, out of all the types of automated test, UI-based tests should constitute the fewest. Why is that?

As suggested above, things take much longer when driving the user interface. Instead, we should test most of our voluminous bits of logic using fast-feedback unit tests. If

the business needs visibility to these permutations and combinations, we can expose them via BDD tests that interact instead with an API or even with a function directly. Save the automated UI tests for verifying end-to-end concerns: Does the application flow properly? Are things wired properly end to end?

User interface tests are brittle and unpredictable. They break easily as things change in the UI, which usually is a frequent occurrence. They fail spuriously due to odd timing or timeout issues; every such failing test wastes our time in ascertaining whether there's a real problem.

When our UI design changes but the system's behavior does not, we incur additional effort to keep the tests running. Either we must update the "glue" code, if we've written the tests declaratively (like we should), or we must update the tests themselves, if our tests are imperative.

Thousands of Tests—Really?

My customer sought to deliver (internally, at least) new functionality at the completion of each two-week iteration. Their stories effectively represented units of work: "Add a field on the policy screen to capture the customer's pet's email address."

To verify that the stories were completed each iteration, the test folk insisted that new scenarios were added for each story: "Verify that the policy screen contains the pet's email address." In other words, the test executed via the scenario required navigating through the application and asserting that the email address could be found on a certain screen. Never mind that there might already have been a scenario that verified information saved with a policy.

One story, one-plus test: This is a recipe for a messy test explosion. This is how you get to ten times the tests you really need. That they happened to also slowly drive the UI only made things another order of magnitude worse.

Your features and scenarios must be groomed carefully over time, even more so than your backlog. If you think of each test as a specification by example, you start to think of the collection of tests as comprehensive documentation on what your system does. And the more you seek to use it as documentation, the more likely you'll organize it well, keep it clean, and keep it meaningful.

How to best organize your tests is a topic for another post. For now, the best thing you can consider is that each scenario should describe how a user accomplishes an end goal in your system. Viewing a field on a screen might be an occasional end goal.

More often, populating that field is a necessary element toward accomplishing some other interesting end goal.

A story represents some unit of work you accomplish. Not every story represents a new end goal. You should frequently be updating scenarios, not always adding new ones.

Stay in Touch!

Make sure you keep up to date with the latest news, learning and events in the Software Testing community by checking out our blog posts and articles on the TestRail blog!

Like the Content? Want More?
Sign up for our weekly Testers Email here on our blog!

SIGN UP HERE!


TestRail
http://www.gurock.com/testrail