RICHARD LAWRENCE

with PAUL RAYNER

Behavior-Driven Development *with* Cucumber

Better Collaboration for Better Software

Behavior-Driven Development with Cucumber

JANE: Well, I see several scenarios that describe the feature we're discussing, the way we want the feature to behave, and each line in the scenario is a step in verifying the behavior for that scenario. That's just like my test plans, which I usually write in Word documents.

I also tend to put in placeholders for tests until I can figure out exactly what the right steps for verifying the correct behavior should be, just like we've done here.

I usually put in specific examples in my test plans too, though I wouldn't normally talk about that with anyone else like we have for this file. I also put in a lot of details about what needs to be clicked in the UI so I remember what I need to do each time I manually step through it.

This is just a plain text file, right?

JONAH: Yes, correct.

The big difference with this, as you said, is that we're doing it now, prior to starting coding. We collaborate together on the examples to use for the feature file, and then we create it *before* Raj and Robin start coding up the feature.

I tend to think of a feature file more as a *collaboration point*, as a springboard for conversation and discovery. The test automation that comes later is a nice side effect of growing these feature files as a kind of "living documentation."

JANE: Does this mean that my test plans go away completely? I'll still do some manual testing, right?

JONAH: Teams doing BDD typically focus the manual testing they do on exploratory testing, on testing the things that fall outside of what the feature file covers.

Since you do test plans now, I suggest you continue following your current practice if it's working for you. Once you get used to using feature files more, I suspect you'll see areas in which you can make improvements to how you approach your testing. That's something to reflect on in future retros.

JANE: Makes sense. It will be great to have more time for exploratory testing.

MARK: Seems straightforward enough to me, too.

JANE: Yes, I'll be interested in seeing where it goes from here. Seems like this approach should reduce a lot of the monkey work I have to do and push the testing much earlier into each sprint, since we'll be doing a lot of it as we go.

ROBIN: I'm confused, so you're saying a feature file is a *test plan*? I thought the feature file was the way we specify how a feature should work. Or am I missing something here?

JONAH: A feature file is not a test plan. What I meant was that some of the things Jane's current test plans accomplish will now happen via running the feature files in Cucumber.

You are right: A feature file specifies how a feature should work, so it functions as an executable specification for the feature so you know what to code. It's more than that, though. It also assumes some responsibility for checking that, after you implement the production code, the feature does actually work the way it should. This is the test automation piece, which we'll be getting to later, once we have a feature file we can automate.

SAM: I get it, you're saying we should get all our feature files formalized in the first step, then do the test automation as a later step. Makes sense to me.

JONAH: I can see how you might get that from what I'm saying, but I'm not talking about treating exploration, formalization, and automation in BDD as separate phases. They are just steps each feature goes through. Sam, it seems to me you're thinking of it as a linear approach, but it's not.

What I am saying is that it's helpful to have a different *mindset* and approach for each step when doing BDD. When exploring examples, you're in a discovery mindset, trying to understand the domain and uncover what assumptions you might have that are incorrect. When formalizing examples, you need to start introducing some rigor around terminology, around the business language you use. When writing feature files, you're trying to drive out ambiguity and incidental details and make sure you're all on the same page with how each feature should work. This will become clearer as we dig into it more.

SAM: I'm not sure I get it but, as you said, let's keep going and hopefully it will be a bit clearer to me.

ROBIN: Jonah, I think I get it. One other question: Should we start using Cucumber for all our automated testing? I've talked to a lot of developers who tried Cucumber and gave up on it. They said the specs were too brittle and slow, and there was just too much overhead. I'm not wanting to be overly negative here, but these are smart people who seemed to know what they were talking about.

JONAH: It's true, a lot of teams try Cucumber and give up on it for the exact reasons you describe. There's a lot I could say about that, but maybe it's sufficient for now to say I'm pretty sure they were trying to use Cucumber just as an automated test runner rather than as a collaboration tool. They were probably writing scenarios for things that would be better handled with their regular unit testing framework, like RSpec or JUnit. Cucumber does introduce overhead with the feature files, but that overhead is worth it when you use the tool for the purposes for which it was intended. It's first and foremost a collaboration tool.

It's hard to explain these things until we actually get into them. These are common points of confusion, so don't lose heart.

There's a model I like for thinking about the different types of testing activities Agile teams do. I've found it can be helpful to see where BDD fits in to the rest of what you do.

Feature Files as Collaboration Points

As we begin formalizing examples, BDD begins to look more like testing. Let's see how it relates to other testing activities. In *Agile Testing*, Lisa Crispin and Janet Gregory described a model for thinking about the various kinds of testing in Agile software development, which they called the "Agile Testing Quadrants," shown in Figure 3-1.

The vertical axis in this model describes whose perspective is taken in the tests. Business-facing tests are tests from the business or customer perspective. They ask questions about whether the system does the functions it ought to do. Technology-facing tests are tests from the developer or team perspective. They ask questions about the implementation: Does the code do what developers intend? Does the system perform well? Is it secure? Of course, customers care about things like performance, but they rarely understand performance tests, and the tests often depend on knowledge of implementation details.

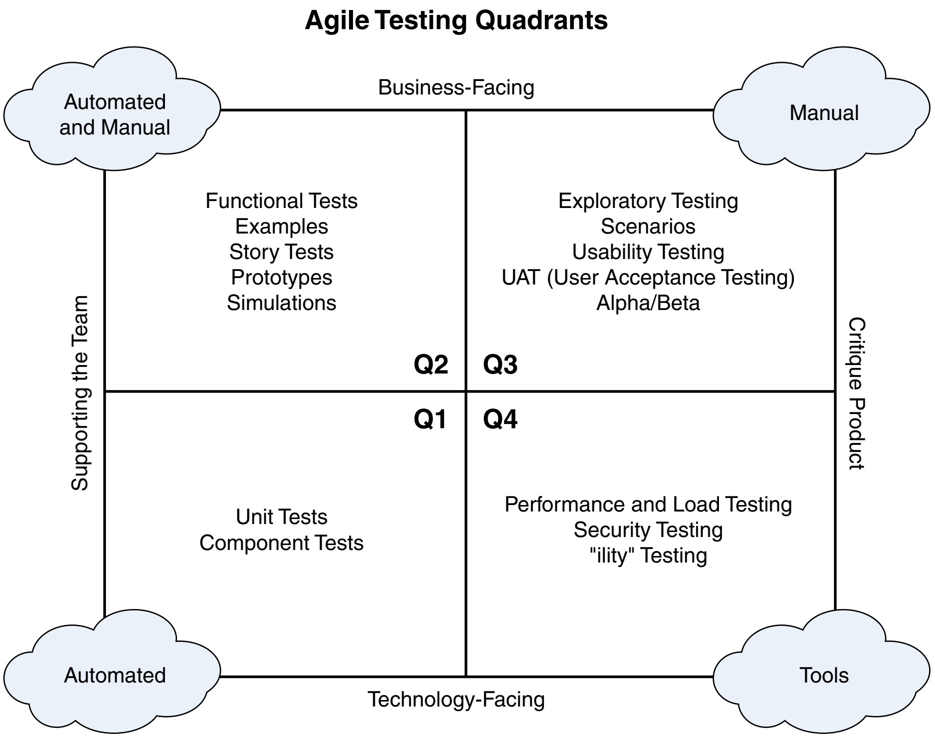

Figure 3-1 *The Agile Testing Quadrants*

The horizontal axis in the model describes when tests are executed relative to development and what function they serve. Tests that critique the product look at something that already exists in order to ask questions about what it does or how it does it. This is what we traditionally think of as testing. On the other side, tests that support the team are tests used to drive development. These tests are written before the software that will make them pass. They act as specifications for features and code.

As we saw in the dialogue, creating an initial draft of a feature file is the first step toward formalizing the examples harvested from the team conversations. As the team talks about the business domain, they seek to understand what the behavior of the system should be in response to the customer's needs. They distill these examples into scenarios in the feature file.

Such examples provide structure and focus to specification, but that's only half the story: Those examples become test scenarios so the team can get automated feedback about the state of the system.

Let's look at how BDD with Cucumber relates to other kinds of testing an Agile team might do based on the Agile Testing Quadrants model.

- **Quadrant 1: Technology-facing tests that support the team:** Quadrant 1 contains technology-facing tests that support the team as it develops the software. This is what we typically think of as test-driven development, or TDD. In TDD, a developer writes a small unit test describing the next bit of code she'd like to create. She runs the test and confirms that it fails, that it describes something not true of the code right now. Then, she writes just enough code to make the test pass. Assuming all previous tests continue to pass, she either refactors the code to improve its design without changing its behavior or writes the next failing unit test. Quadrant 1 tests are automated; they're run many, many times in the course of development and must be fast and easy to use.

  In the course of making a single Cucumber scenario pass, a developer using TDD will typically write many unit tests. TDD is a small loop inside the bigger BDD loop.
- **Quadrant 4: Technology-facing tests that critique the product:** Quadrant 4 tests cover what are typically referred to as "nonfunctional requirements," such as performance, security, scalability, and so on. These tests ask questions about a product or increment of product that already has been developed. Often, such tests require a dedicated, production-like test environment, making them difficult to execute during development. Nonetheless, teams should find ways to get this feedback early and often lest they discover a problem too late to do anything about it. Quadrant 4 tests may be automated. Even when they're not automated, they're typically tool-assisted.

  Quadrant 4 tests usually address different concerns from Cucumber scenarios. Together, they describe the system more fully.
- **Quadrant 3: Business-facing tests that critique the product:** This is the work typically done by people with "tester" or "QA" in their job title. They look at a product or product increment that already exists and ask questions about whether it does what it ought to do. They look for edge cases, boundary conditions, and defects. Quadrant 3 tests tend to be manual. They ask questions that were not anticipated by automated tests.

  Failing Quadrant 3 tests often become new Cucumber scenarios to drive the necessary development to make those tests pass.

- **Quadrant 2: Business-facing tests that support the team:** BDD fits in Quadrant 2. In Quadrant 2, tests address the behavior of the system from the business perspective. Unlike Quadrant 3 tests, however, Quadrant 2 tests are written *before* the code that makes them pass, and thus are used as a tool to write that code correctly. Quadrant 2 tests are executable examples acting as functional specifications.

Mapping Your Current Testing Activities

Consider the testing activities your team currently does. How do they map to the quadrants? Do you do any testing to support the team, or do all your tests critique the product? If your developers write unit tests but not until after the code exists, you don't do Quadrant 1 tests—those unit tests aren't supporting the team during development. If you're like many Agile teams, you have decent coverage of Quadrant 3, some activity in Quadrant 4, occasional Quadrant 1, and nothing in Quadrant 2.

How Will Adopting BDD Affect Your Testing?

By adopting BDD, you will fill in more of Quadrant 2, allowing testers to do more interesting tests in Quadrant 3 and perhaps dedicate more time to Quadrant 4. In the process, you'll be investing in creating a foundation and motivation for the adoption of developer-facing TDD to fill in Quadrant 1.

Quadrant 2 tests can create a foundation for moving into Quadrant 1. They build a safety net for the refactoring most teams need to do to make their code more testable in small units. They teach the test-first approach in a way that's easy to understand, and they create motivation for more attention to be paid to Quadrant 1 because Quadrant 2 tests are inevitably larger and slower. Once teams become accustomed to automated tests supporting their development, they want faster, more granular tests. Eventually, each Quadrant 2 test will drive the development of multiple Quadrant 1 tests as part of developing the code.

In the dialogues, our library team is currently working on Quadrant 2 tests with Cucumber. The scenarios so far test the new ebook search behavior from the user's perspective. In the next chapter, they'll begin automating these scenarios. The automation will drive the library website, which is realistic but relatively slow. Raj and Robin might decide they want faster feedback on a part of the ebook search engine as they develop it. This will motivate them to create smaller, faster unit tests for themselves (Quadrant 1).

Growing Quadrant 2 Collaboration Capabilities

A feature file is a Quadrant 2 collaboration point; a springboard for conversation and discovery. The Quadrant 2 test automation that comes later is a nice side effect