The Addison-Wesley Signature Series

A Mike Cohn Signature Book

Agile Testing

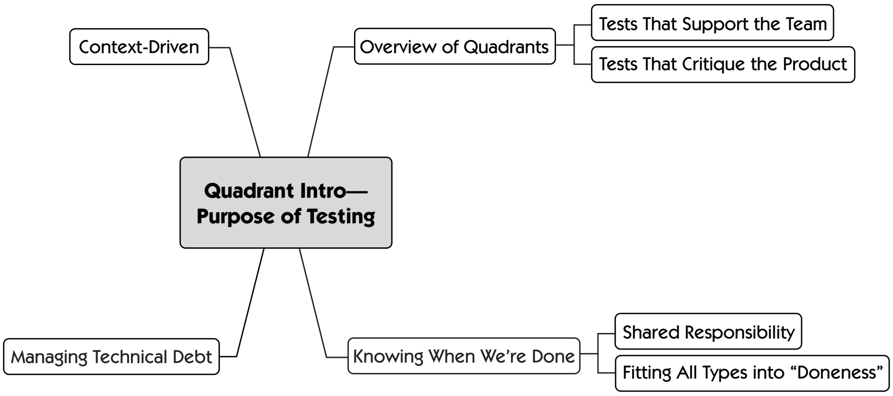
A Practical Guide for Testers and Agile Teams

Lisa Crispin
Janet Gregory

Forewords by Mike Cohn and Brian Marick

Chapter 6

THE PURPOSE OF TESTING


Context-Driven
Overview of Quadrants
Tests That Support the Team
Tests That Critique the Product
Quadrant Intro—Purpose of Testing
Managing Technical Debt
Knowing When We're Done
Shared Responsibility
Fitting All Types into "Doneness"

Why do we test? The answer might seem obvious, but in fact, it's pretty complex. We test for a lot of reasons: to find bugs, to make sure the code is reliable, and sometimes just to see if the code's usable. We do different types of testing to accomplish different goals. Software product quality has many components. In this chapter, we introduce the Agile Testing Quadrants. The rest of the chapters in Part III go into detail on each of the quadrants. The Agile Testing Quadrants matrix helps testers ensure that they have considered all of the different types of tests that are needed in order to deliver value.

THE AGILE TESTING QUADRANTS

In Chapter 1, "What Is Agile Testing, Anyway?," we introduced Brian Marick's terms for different categories of tests that accomplish different purposes. Figure 6-1 is a diagram of the agile testing quadrants that shows how each of the four quadrants reflects the different reasons we test. On one axis, we divide the matrix into tests that support the team and tests that critique the product. The other axis divides them into business-facing and technology-facing tests.

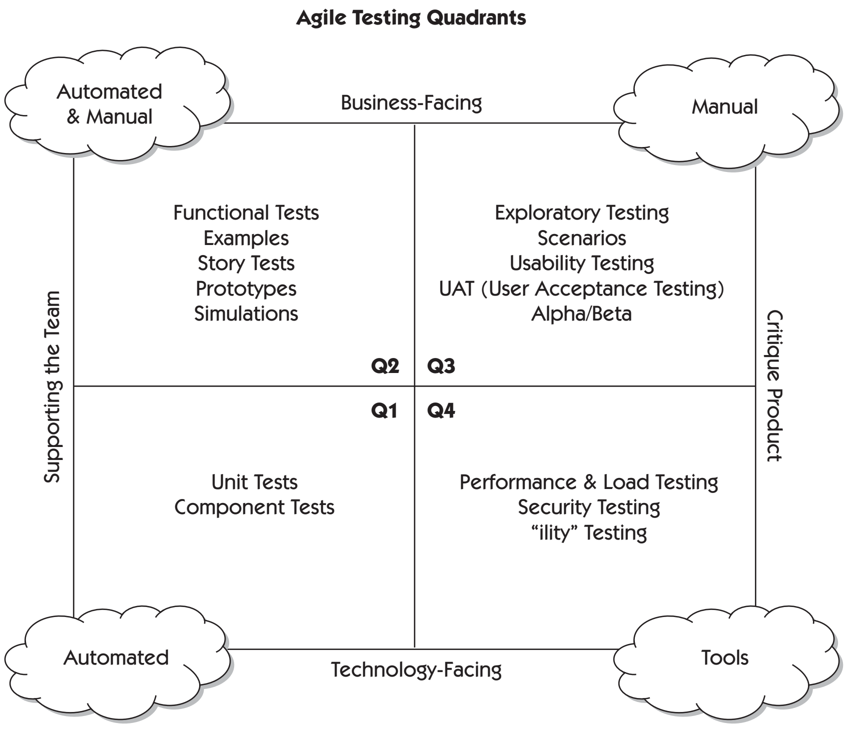


Figure 6-1 Agile Testing Quadrants

The order in which we've numbered these quadrants has no relationship to when the different types of testing are done. For example, agile development starts with customer tests, which tell the team what to code. The timing of the various types of tests depends on the risks of each project, the customers' goals for the product, whether the team is working with legacy code or on a greenfield project, and when resources are available to do the testing.

Tests that Support the Team

The quadrants on the left include tests that support the team as it develops the product. This concept of testing to help the programmers is new to many testers and is the biggest difference between testing on a traditional project and testing on an agile project. The testing done in Quadrants 1 and 2 are more requirements specification and design aids than what we typically think of as testing.

Quadrant 1

The lower left quadrant represents test-driven development, which is a core agile development practice.

Unit tests verify functionality of a small subset of the system, such as an object or method. Component tests verify the behavior of a larger part of the system, such as a group of classes that provide some service [Meszaros, 2007]. Both types of tests are usually automated with a member of the xUnit family of test automation tools. We refer to these tests as programmer tests, developer-facing tests, or technology-facing tests. They enable the programmers to measure what Kent Beck has called the internal quality of their code [Beck, 1999].

A major purpose of Quadrant 1 tests is test-driven development (TDD) or test-driven design. The process of writing tests first helps programmers design their code well. These tests let the programmers confidently write code to deliver a story's features without worrying about making unintended changes to the system. They can verify that their design and architecture decisions are appropriate. Unit and component tests are automated and written in the same programming language as the application. A business expert probably couldn't understand them by reading them directly, but these tests aren't intended for customer use. In fact, internal quality isn't negotiated with the customer; it's defined by the programmers. Programmer tests are normally part of an automated process that runs with every code check-in, giving the team instant, continual feedback about their internal quality.

Quadrant 2

The tests in Quadrant 2 also support the work of the development team, but at a higher level. These business-facing tests, also called customer-facing tests and customer tests, define external quality and the features that the customers want.

Chapter 8, "Business-Facing Tests that Support the Team," explains business conditions of satisfaction.

Like the Quadrant 1 tests, they also drive development, but at a higher level. With agile development, these tests are derived from examples provided by the customer team. They describe the details of each story. Business-facing tests run at a functional level, each one verifying a business satisfaction condition. They're written in a way business experts can easily understand using the business domain language. In fact, the business experts use these tests to define the external quality of the product and usually help to write them. It's possible this quadrant could duplicate some of the tests that were done at the unit level; however, the Quadrant 2 tests are oriented toward illustrating and confirming desired system behavior at a higher level.

Most of the business-facing tests that support the development team also need to be automated. One of the most important purposes of tests in these two quadrants is to provide information quickly and enable fast troubleshooting. They must be run frequently in order to give the team early feedback in case any behavior changes unexpectedly. When possible, these automated tests run directly on the business logic in the production code without having to go through a presentation layer. Still, some automated tests must verify the user interfaces and any APIs that client applications might use. All of these tests should be run as part of an automated continuous integration, build, and test process.

There is another group of tests that belongs in this quadrant as well. User interaction experts use mock-ups and wireframes to help validate proposed GUI (graphical user interface) designs with customers and to communicate those designs to the developers before they start to code them. The tests in this group are tests that help support the team to get the product built right but are not automated. As we'll see in the following chapters, the quadrants help us identify all of the different types of tests we need to use in order to help drive coding.

Some people use the term "acceptance tests" to describe Quadrant 2 tests, but we believe that acceptance tests encompass a broader range of tests that include Quadrants 3 and 4. Acceptance tests verify that all aspects of the system, including qualities such as usability and performance, meet customer requirements.

Using Tests to Support the Team

The quick feedback provided by Quadrants 1 and 2 automated tests, which run with every code change or addition, form the foundation of an agile team. These tests first guide development of functionality, and when automated, then provide a safety net to prevent refactoring and the introduction of new code from causing unexpected results.

Lisa's Story

We run our automated tests that support the team (the left half of the quadrants) in separate build processes. Unit and component tests run in our "ongoing" build, which takes about eight minutes to finish. Although the programmers run the unit tests before they check in, the build might still fail due to integration problems or environmental differences. As soon as we see the "build failed" email, the person who checked in the offending code fixes the problem. Business-facing functional tests run in our "full build," which also runs continually, kicking off every time a code change is checked in. It finishes in less than two hours. That's still pretty quick feedback, and again, a build failure means immediate action to fix the problem. With these builds as a safety net, our code is stable enough to release every day of the iteration if we so choose.

—Lisa

The tests in Quadrants 1 and 2 are written to help the team deliver the business value requested by the customers. They verify that the business logic and the user interfaces behave according to the examples provided by the customers. There are other aspects to software quality, some of which the customers don't think about without help from the technical team. Is the product competitive? Is the user interface as intuitive as it needs to be? Is the application secure? Are the users happy with how the user interface works? We need different tests to answer these types of questions.

Tests that Critique the Product

If you've been in a customer role and had to express your requirements for a software feature, you know how hard it can be to know exactly what you want until you see it. Even if you're confident about how the feature should work, it can be hard to describe it so that programmers fully understand it.

The word "critique" isn't intended in a negative sense. A critique can include both praise and suggestions for improvement. Appraising a software product involves both art and science. We review the software in a constructive manner, with the goal of learning how we can improve it. As we learn, we can feed new requirements and tests or examples back to the process that supports the team and guide development.

Quadrant 3

Business-facing examples help the team design the desired product, but at least some of our examples will probably be wrong. The business experts might overlook functionality, or not get it quite right if it isn't their field of expertise. The team might simply misunderstand some examples. Even when the programmers write code that makes the business-facing tests pass, they might not be delivering what the customer really wants.

That is where the tests to critique the product in the third and fourth quadrants come into play. Quadrant 3 classifies the business-facing tests that exercise the working software to see if it doesn't quite meet expectations or won't stand up to the competition. When we do business-facing tests to critique the product, we try to emulate the way a real user would work the application. This is manual testing that only a human can do. We might use some automated scripts to help us set up the data we need, but we have to use our senses, our brains, and our intuition to check whether the development team has delivered the business value required by the customers.

Often, the users and customers perform these types of tests. User Acceptance Testing (UAT) gives customers a chance to give new features a good workout and see what changes they may want in the future, and it's a good way to gather new story ideas. If your team is delivering software on a contract basis to a client, UAT might be a required step in approving the finished stories.

Usability testing is an example of a type of testing that has a whole science of its own. Focus groups might be brought in, studied as they use the application, and interviewed in order to gather their reactions. Usability testing can also include navigation from page to page or even something as simple as the tabbing order. Knowledge of how people use systems is an advantage when testing usability.

Exploratory testing is central to this quadrant. During exploratory testing sessions, the tester simultaneously designs and performs tests, using critical thinking to analyze the results. This offers a much better opportunity to learn about the application than scripted tests. We're not talking about ad hoc testing, which is impromptu and improvised. Exploratory testing is a more thoughtful and sophisticated approach than ad hoc testing. It is guided by a strategy and operates within defined constraints. From the start of each project and story, testers start thinking of scenarios they want to try. As small chunks of testable code become available, testers analyze test results, and as they learn, they find new areas to explore. Exploratory testing works the system in the same ways that the end users will. Testers use their creativity and intuition. As a result, it is through this type of testing that many of the most serious bugs are usually found.

Quadrant 4

The types of tests that fall into the fourth quadrant are just as critical to agile development as to any type of software development. These tests are technology-facing, and we discuss them in technical rather than business terms. Technology-facing tests in Quadrant 4 are intended to critique product characteristics such as performance, robustness, and security. As we'll describe in Chapter 11, "Critiquing the Product using Technology-Facing Tests," your team already possesses many of the skills needed to do these tests. For example, programmers might be able to leverage unit tests into performance tests with a multi-threaded engine. However, creating and running these tests might require the use of specialized tools and additional expertise.

In the past, we've heard complaints that agile development seems to ignore the technology-facing tests that critique the product. These complaints might be partly due to agile's emphasis on having customers write and prioritize stories. Nontechnical customer team members often assume that the developers will take care of concerns such as speed and security, and that the programmers are intent on producing only the functionality prioritized by the customers.

If we know the requirements for performance, security, interaction with other systems, and other nonfunctional attributes before we start coding, it's easier to design and code with that in mind. Some of these might be more important than actual functionality. For example, if an Internet retail website has a one-minute response time, the customers won't wait to appreciate the fact that all of the features work properly. Technology-facing tests that critique the product should be considered at every step of the development cycle and not left until the very end. In many cases, such testing should even be done before functional testing.

In recent years we've seen many new lightweight tools appropriate to an agile development project become available to support tests. Automation tools can be used to create test data, set up test scenarios for manual testing, drive security tests, and help make sense of results. Automation is mandatory for some efforts such as load and performance testing.

Checking Nonfunctional Requirements

Alessandro Collino, a computer science and information engineer with Onion S.p.A., who works on agile projects, illustrates why executing tests that critique the product early in the development process is critical to project success.

> Our Scrum/XP team used TDD to develop a Java application that would convert one form of XML to another. The application performed complex calculations on the data. For each simple story, we wrote a unit test to check the conversion of one element into the required format, implemented the code to make the test pass, and refactored as needed.
>
> We also wrote acceptance tests that read subsets of the original XML files from disk, converted them, and wrote them back. The first time we ran the application on a real file to be converted, we got an out-of-memory error. The DOM parser we used for the XML conversion couldn't handle such a large file. All of our tests used small subsets of the actual files; we hadn't thought to write unit tests using large datasets.

> Doing TDD gave us quick feedback on whether the code was working per the functional requirements, but the unit tests didn't test any non-functional requirements such as capacity, performance, scalability, and usability. If you use TDD to also check nonfunctional requirements, in this case, capacity, you'll have quick feedback and be able to avoid expensive mistakes.

Alessandro's story is a good example of how the quadrant numbering doesn't imply the order in which tests are done. When application performance is critical, plan to test with production-level loads as soon as testable code is available.

When you and your team plan a new release or project, discuss which types of tests from Quadrants 3 and 4 you need, and when they should be done. Don't leave essential activities such as load or usability testing to the end, when it might be too late to rectify problems.

Using Tests that Critique the Product

The information produced during testing to review the product should be fed back into the left side of our matrix and used to create new tests to drive future development. For example, if the server fails under a normal load, new stories and tests to drive a more scalable architecture will be needed. Using the quadrants will help you plan tests that critique the product as well as tests that drive development. Think about why you are testing to make sure that the tests are performed at the optimum stage of development.

The short iterations of agile development give your team a chance to learn and experiment with the different testing quadrants. If you find out too late that your design doesn't scale, start load testing earlier with the next story or project. If the iteration demo reveals that the team misunderstood the customer's requirements, maybe you're not doing a good enough job of writing customer tests to guide development. If the team puts off needed refactoring, maybe the unit and component tests aren't providing enough coverage. Use the agile testing quadrants to help make sure all necessary testing is done at the right time.

Knowing When a Story Is Done

For most products, we need all four categories of testing to feel confident we're delivering the right value. Not every story requires security testing, but you don't want to omit it because you didn't think of it.

Lisa's Story

My team uses "stock" cards to ensure that we always consider all different types of tests. When unit testing wasn't yet a habit, we wrote a unit test card for each story on the board. Our "end to end" test card reminds the programmers to complete the job of integration testing and to make sure all of the parts of the code work together. A "security" card also gets considered for each story, and if appropriate, put on the board to keep everyone conscious of keeping data safe. A task card to show the user interface to customers makes sure that we don't forget to do this as early as possible, and it helps us start exploratory testing along with the customers early, too. All of these cards help us address all the different aspects of product quality.

Technology-facing tests that extend beyond a single story get their own row on the story board. We use stories to evaluate load test tools and to establish performance baselines to kick off our load and performance-testing efforts.

—Lisa

The technology-facing and business-facing tests that drive development are central to agile development, whether or not you actually write task cards for them. They give your team the best chance of getting each story "done." Identifying the tasks needed to perform the technology-facing and business-facing tests that critique the product ensures that you'll learn what the product is missing. A combination of tests from all four quadrants will let the team know when each feature has met the customer's criteria for functionality and quality.

Shared Responsibility

Our product teams need a wide range of expertise to cover all of the agile testing quadrants. Programmers should write the technology-facing tests that support programming, but they might need help at different times from testers, database designers, system administrators, and configuration specialists. Testers take primary charge of the business-facing tests in tandem with the customers, but programmers participate in designing and automating tests, while usability and other experts might be called in as needed. The fourth quadrant, with technology-facing tests that critique the product, may require more specialists. No matter what resources have to be brought in from outside the development team, the team is still responsible for getting all four quadrants of testing done.

We believe that a successful team is one where everybody participates in the crafting of the product and that everyone shares the team's internal pain when things go wrong. Implementing the practices and tools that enable us to address all four quadrants of testing can be painful at times, but the joy of implementing a successful product is worth the effort.

MANAGING TECHNICAL DEBT

Ward Cunningham coined the term "technical debt" in 1992, but we've certainly experienced it throughout our careers in software development! Technical debt builds up when the development team takes shortcuts, hacks in quick fixes, or skips writing or automating tests because it's under the gun. The code base gets harder and harder to maintain. Like financial debt, "interest" compounds in the form of higher maintenance costs and lower team velocity. Programmers are afraid to make any changes, much less attempt refactoring to improve the code, for fear of breaking it. Sometimes this fear exists because they can't understand the coding to start with, and sometimes it is because there are no tests to catch mistakes.

Each quadrant in the agile testing matrix plays a role in keeping technical debt to a manageable level. Technology-facing tests that support coding and design help keep code maintainable. An automated build and integration process that runs unit tests is a must for minimizing technical debt. Catching unit-level defects during coding will free testers to focus on business-facing tests in order to guide the team and improve the product. Timely load and stress testing lets the teams know whether their architecture is up to the job.

By taking the time and applying resources and practices to keep technical debt to a minimum, a team will have time and resources to cover the testing needed to ensure a quality product. Applying agile principles to do a good job of each type of testing at each level will, in turn, minimize technical debt.

TESTING IN CONTEXT

Categorizations and definitions such as we find in the agile testing matrix help us make sure we plan for and accomplish all of the different types of testing we need. However, we need to bear in mind that each organization, product, and team has its own unique situation, and each needs to do what works for it in its individual situation. As Lisa's coworker Mike Busse likes to say, "It's a tool, not a rule." A single product or project's needs might evolve drastically over time. The quadrants are a helpful way to make sure your team is considering all of the different aspects of testing that go into "doneness."

We can borrow important principles from the context-driven school of testing when planning testing for each story, iteration, and release.

For more on context-driven testing, see www.context-driven-testing.com.

- The value of any practice depends on its context.
- There are good practices in context, but there are no best practices.
- People, working together, are the most important part of any project's context.
- Projects unfold over time in ways that are often not predictable.
- The product is a solution. If the problem isn't solved, the product doesn't work.
- Good software testing is a challenging intellectual process.
- Only through judgment and skill, exercised cooperatively throughout the entire project, are we able to do the right things at the right times to effectively test our products.

The quadrants help give context to agile testing practices, but you and your team will have to adapt as you go. Testers help provide the feedback the team needs to adjust and work better. Use your skills to engage the customers throughout each iteration and release. Be conscious of when your team needs roles or knowledge beyond what it currently has available.

The Agile Testing Quadrants provide a checklist to make sure you've covered all your testing bases. Examine the answers to questions such as these:

- Are we using unit and component tests to help us find the right design for our application?
- Do we have an automated build process that runs our automated unit tests for quick feedback?
- Do our business-facing tests help us deliver a product that matches customers' expectations?
- Are we capturing the right examples of desired system behavior? Do we need more? Are we basing our tests on these examples?
- Do we show prototypes of UIs and reports to the users before we start coding them? Can the users relate them to how the finished software will work?
- Do we budget enough time for exploratory testing? How do we tackle usability testing? Are we involving our customers enough?
- Do we consider technological requirements such as performance and security early enough in the development cycle? Do we have the right tools to do "ility" testing?

Use the matrix as a map to get started. Experiment, and use retrospectives to keep improving your efforts to guide development with tests and build on what you learn about your product through testing.

Summary

In this chapter we introduced the Agile Testing Quadrants as a convenient way to categorize tests. The four quadrants serve as guidelines to ensure that all facets of product quality are covered in the testing and developing process.

- Tests that support the team can be used to drive requirements.
- Tests that critique the product help us think about all facets of application quality.
- Use the quadrants to know when you're done, and ensure the whole team shares responsibility for covering the four quadrants of the matrix.
- Managing technical debt is an essential foundation for any software development team. Use the quadrants to think about the different dimensions.
- Context should always guide our testing efforts.