Part III

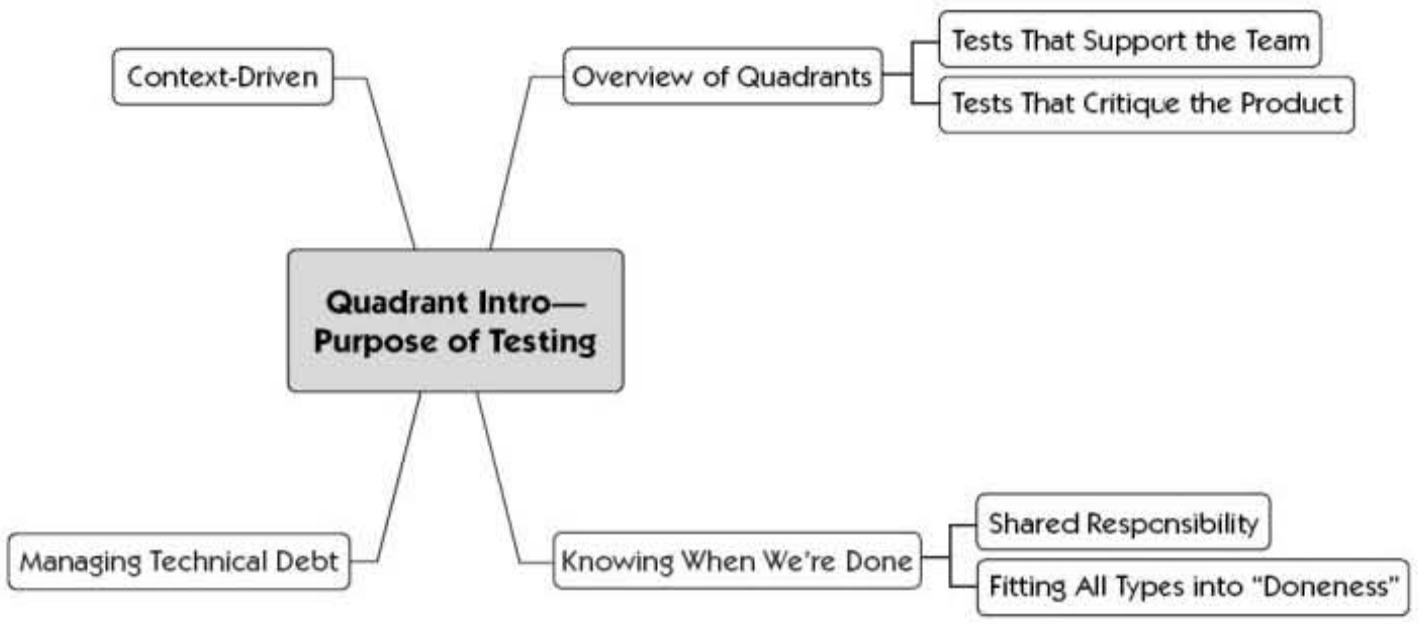
The Agile Testing Quadrants

Software quality has many dimensions, each requiring a different testing approach. How do we know all the different types of tests we need to do? How do we know when we're "done" testing? Who does which tests and how? In this part, we explain how to use the Agile Testing Quadrants to make sure your team covers all needed categories of testing.

Of course, testing requires tools, and we've included examples of tools to use, strategies for using those tools effectively, and guidelines about when to use them. Tools are easier to use when used with code that's designed for testability. These concerns and more are discussed in this part of the book.

Chapter 6

THE PURPOSE OF TESTING



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.

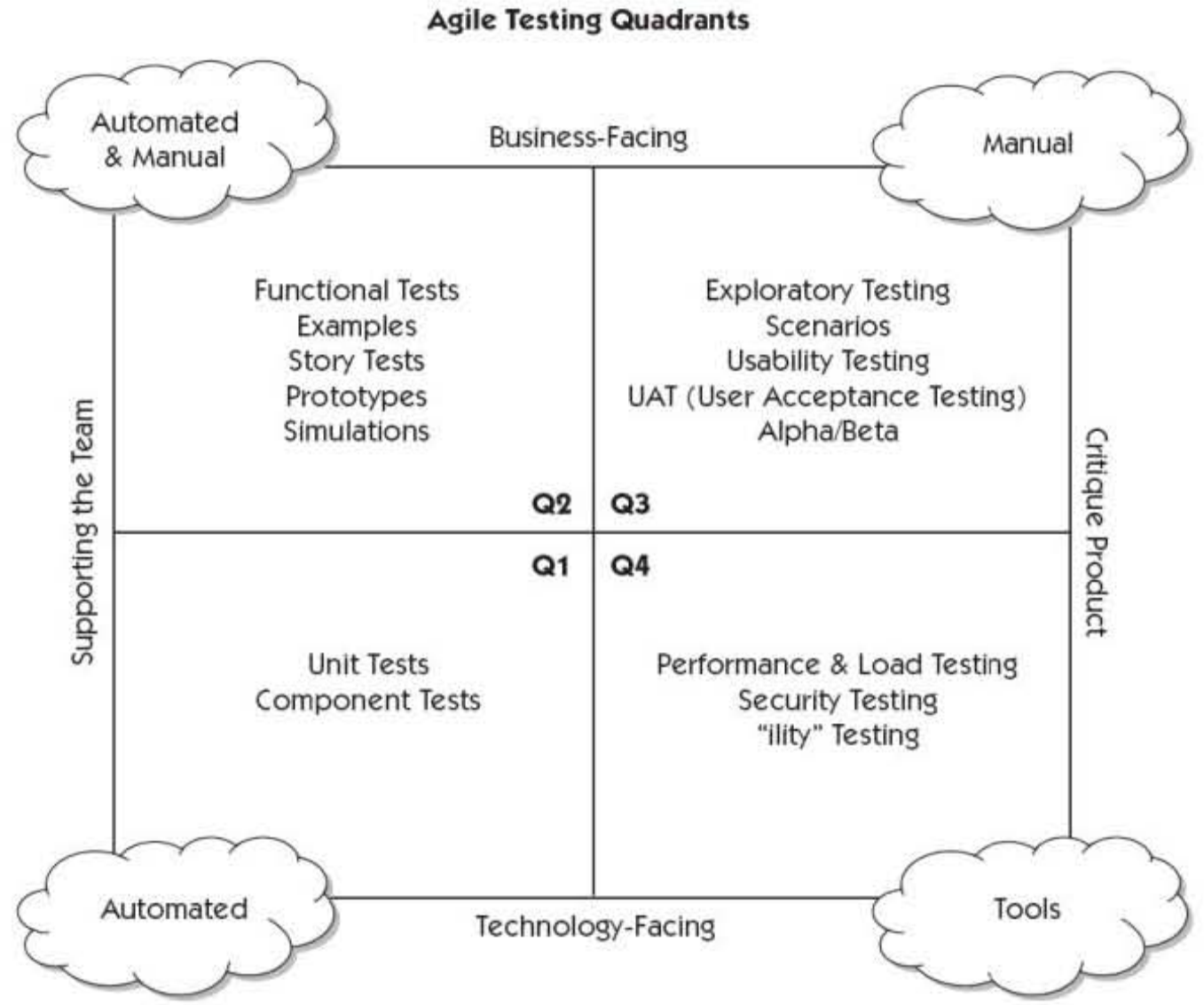


Figure 6-1 Agile Testing Quadrants

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 trouble-shooting. 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 nonfunctional 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.

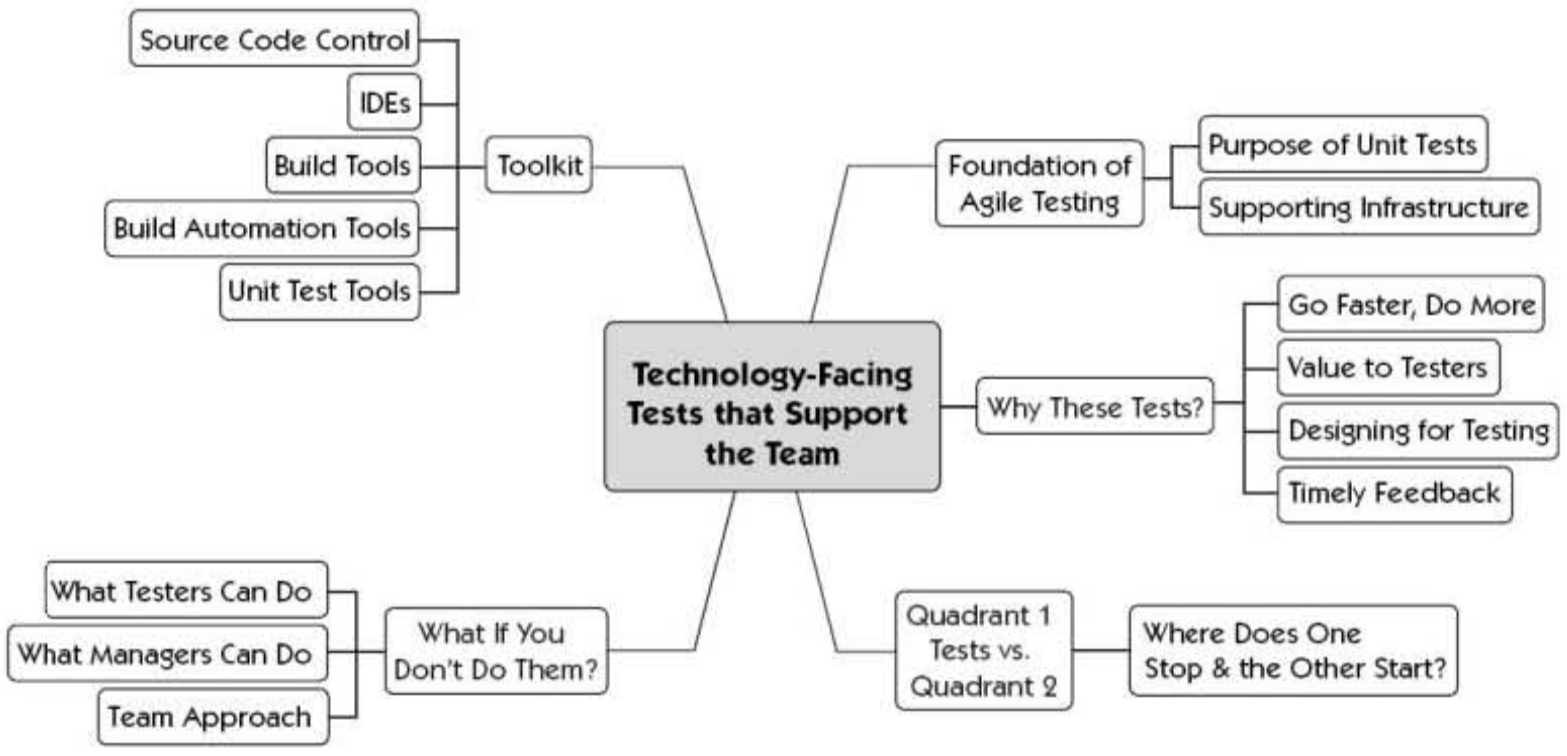
SUMMARY

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.

Chapter 7

TECHNOLOGY-FACING TESTS THAT SUPPORT THE TEAM



We use the Agile Testing Quadrants as a guide to help us cover all the types of testing we need and to help us make sure we have the right resources to succeed at each type. In this chapter, we look at tests in the first quadrant, technology-facing tests that support the team, and at tools to support this testing. The activities in this quadrant form the core of agile development.

AN AGILE TESTING FOUNDATION

We discuss Quadrant 1 first because the technology-facing tests that support the team form the foundation of agile development and testing. See Figure 7-1 for a reminder of the Agile Testing Quadrants with this quadrant highlighted. Quadrant 1 is about much more than testing. The unit and component tests we talk about in Quadrant 1 aren't the first tests written for each story, but they help guide design and development. Without a foundation of test-driven

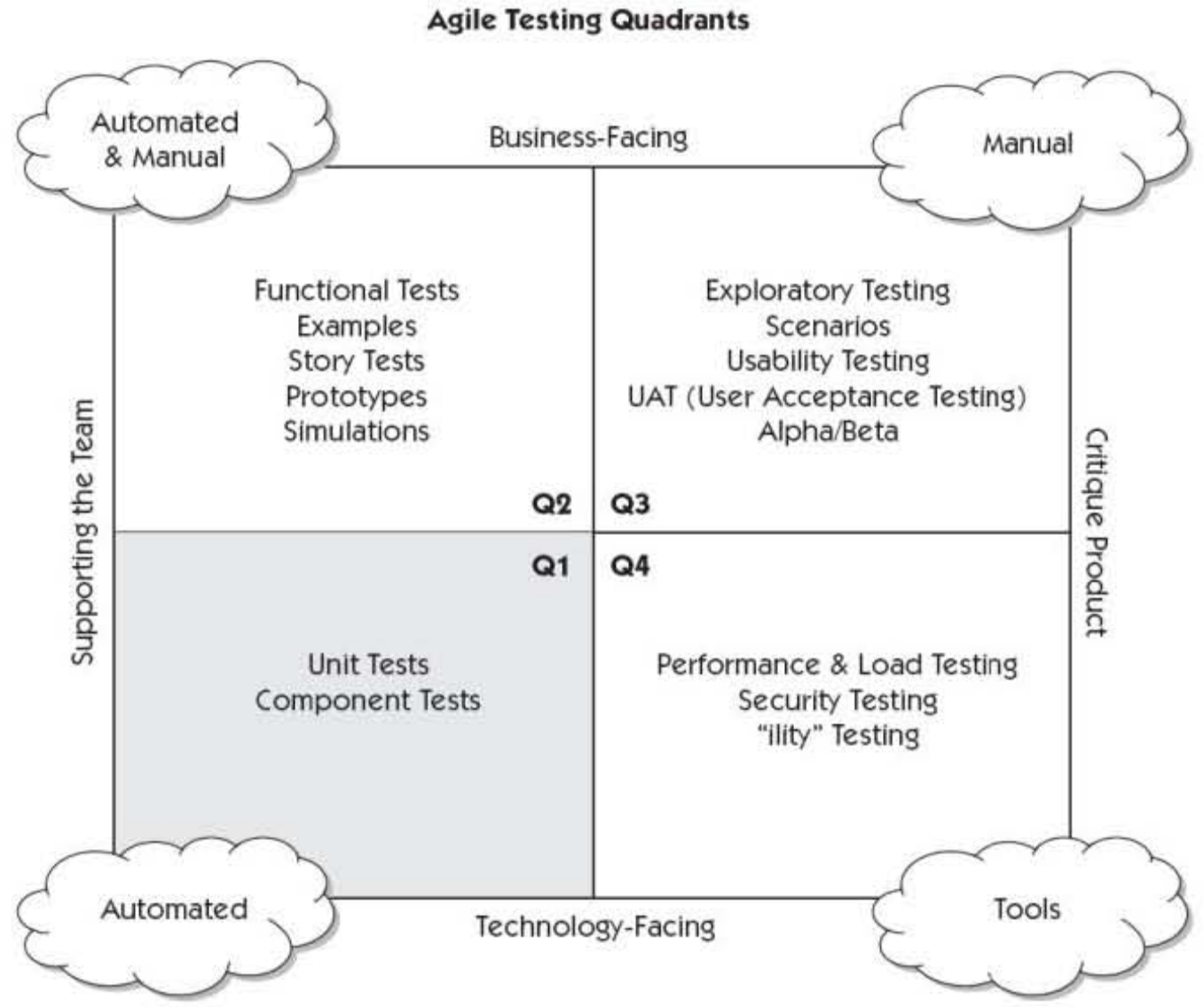


Figure 7-1 The Agile Testing Quadrants, highlighting Quadrant 1

design, automated unit and component tests, and a continuous integration process to run the tests, it's hard to deliver value in a timely manner. All of the testing in the other quadrants can't make up for inadequacies in this one. We'll talk about the other quadrants in the next few chapters and explain how they all fit together.

Teams need the right tools and processes to create and execute technology-facing tests that guide development. We'll give some examples of the types of tools needed in the last section of this chapter.

The Purpose of Quadrant 1 Tests

Unit tests and component tests ensure quality by helping the programmer understand exactly what the code needs to do, and by providing guidance in

the right design. They help the team to focus on the story that's being delivered and to take the simplest approach that will work. Unit tests verify the behavior of parts as small as a single object or method [Meszaros, 2007]. Component tests help solidify the overall design of a deployable part of the system by testing the interaction between classes or methods.

Developing unit tests can be an essential design tool when using TDD. When an agile programmer starts a coding task, she writes a test that captures the behavior of a tiny bit of code and then works on the code until the test passes. By building the code in small test-code-test increments, the programmer has a chance to think through the functionality that the customer needs. As questions come up, she can ask the customer. She can pair with a tester to help make sure all aspects of that piece of code, and its communication with other units, are tested.

The term *test-driven development* misleads practitioners who don't understand that it's more about design than testing. Code developed test-first is naturally designed for testability. Quadrant 1 activities are all aimed at producing software with the highest possible internal quality.

When teams practice TDD, they minimize the number of bugs that have to be caught later on. Most unit-level bugs are prevented by writing the test before the code. Thinking through the design by writing the unit test means the system is more likely to meet customer requirements. When post-development testing time is occupied with finding and fixing bugs that could have been detected by programmer tests, there's no time to find the serious issues that might adversely affect the business. The more bugs that leak out of our coding process, the slower our delivery will be, and in the end, it is the quality that will suffer. That's why the programmer tests in Quadrant 1 are so critical. While every team should adopt practices that work for its situation, a team without these core agile practices is unlikely to benefit much from agile values and principles.

Supporting Infrastructure

Solid source code control, configuration management, and continuous integration are essential to getting value from programmer tests that guide development. They enable the team to always know exactly what's being tested. Continuous integration gives us a way to run tests every time new code is checked in. When a test fails, we know who checked in the change that caused the failure, and that person can quickly fix the problem. Continuous

integration saves time and motivates each programmer to run the tests before checking in the new code. A continuous integration and build process delivers a deployable package of code for us to test.

Agile projects that lack these core agile practices tend to turn into "mini-waterfalls." The development cycles are shorter, but code is still being thrown "over the wall" to testers who run out of time to test because the code is of poor quality. The term *waterfall* isn't necessarily derogatory. We've worked on successful "waterfall" projects where the programmers stringently automate unit tests, practice continuous integration, and use automated builds to run tests. These successful "waterfall" projects also involve customers and testers throughout the development cycle. When we code without appropriate practices and tools, regardless of what we call the process, we're not going to deliver high-quality code in a timely manner.

WHY WRITE AND EXECUTE THESE TESTS?

We're not going into any details here about how to do TDD, or the best ways to write unit and component tests. There are several excellent books on those subjects. Our goal is to explain why these activities are important to agile testers. Let's explore some reasons to use technology-facing tests that support the team.

Lets Us Go Faster and Do More

Speed should never be the end goal of an agile development team. Trying to do things fast and meet tight deadlines without thinking about the quality causes us to cut corners and revert to old, bad habits. If we cut corners, we'll build up more technical debt, and probably miss the deadline anyway. Happily, though, speed is a long-term side effect of producing code with the highest possible internal quality. Continuous builds running unit tests notify the team of failure within a few minutes of the problem check-in, and the mistake can be found and fixed quickly. A safety net of automated unit and code integration tests enables the programmers to refactor frequently. This keeps the code at a reasonable standard of maintainability and delivers the best value for the time invested. Technical debt is kept as low as possible.

If you've worked as a tester on a project where unit testing was neglected, you know how easy it is to spend all of your time finding unit-level defects. You might find so many bugs while testing the "happy path" that you never have time to test more complex scenarios and edge cases. The release deadline is pushed back as the "find and fix" cycle drags on, or testing is just stopped and a buggy product is foisted off on unsuspecting customers.

Our years on agile teams have been Utopian in contrast to this scenario. Driving coding practices with tests means that the programmers probably understood the story's requirements reasonably well. They've talked extensively with the customers and testers to clarify the desired behaviors. All parties understand the changes being made. By the time the team has completed all of the task cards for coding a story, or a thin, testable slice of one, the feature has been well covered by unit and component tests. Usually the programmers have made sure at least one path through the story works end to end.

This means that we, as testers, waste little time finding low-level bugs. We're likely to try scenarios the programmers hadn't thought of and to spend our time on higher-level business functionality. Well-designed code is usually robust and testable. If we find a defect, we show it to the programmer, who writes a unit test to reproduce the bug and then fixes it quickly. We actually have time to focus on exploratory testing and the other types of in-depth tests to give the code a good workout and learn more about how it should work. Often, the only "bugs" we find are requirements that everyone on our team missed or misunderstood. Even those are found quickly if the customer is involved and has regular demos and test opportunities. After a development team has mastered TDD, the focus for improvement shifts from bug prevention to figuring out better ways to elicit and capture requirements before coding.

Test-First Development vs. Test-Driven Development

Gerard Meszaros [Meszaros 2007, pp. 813–814] offers the following description of how test-first development differs from test-driven development:

> "Unlike test-driven development, test-first development merely says that the tests are written before the production code; it does not imply that the production code is made to work one test at a time (emergent design). Test-first development can be applied at the unit test or customer test level, depending on which tests we have chosen to automate."

Erik Bos [2008] observes that test-first development involves both test-first programming and test-first design, but there's a subtle difference:

> "With test-first design, the design follows the tests, whereas you can do test-first programming of a design that you first write down on a whiteboard. On larger projects, we tend to do more design via whiteboard discussions; the team discusses the architecture around a whiteboard, and codes test-first based on this design. On smaller projects, we do practice test-driven design."

There are several different philosophies about when to write tests and for what purpose. It's up to each team to agree on the approach that helps it achieve its quality objectives, although there is common agreement in the agile community that TDD definitely helps a team achieve better-quality software. This is an important way that programmer tests support the team. Let's look at some more ways.

Making Testers' Jobs Easier

The core practices related to programmer tests make lots of testing activities easier to accomplish. Programmers work in their own sandboxes, where they can test new code without affecting anyone else's work. They don't check in code until it has passed a suite of regression tests in their sandbox.

The team thinks about test environments and what to use for test data. Unit tests usually work with fake or mock objects instead of actual databases for speed, but programmers still need to test against realistic data. Testers can help them identify good test data. If the unit tests represent real-life data, fewer issues will be found later.

Lisa's Story

Here's a small example. When my current team first adopted agile development, we didn't have any automated tests. We had no way to produce a deployable code package, and we had no rudimentary test environments or test databases. I didn't have any means to produce a build myself, either. We decided to start writing code test-first and committed to automating tests at all levels where appropriate, but we needed some infrastructure first.

Our first priority was to implement a continuous build process, which was done in a couple of days. Each build sent an email with a list of checked-in files and comments about the updates. I could now choose which build to deploy and test. The next priority was to provide independent test environments so that tests run by one person would not interfere with other tests. The new database expert created new schemas to meet testing needs and a "seed" database of canonical, production-like data. These schemas could be refreshed on demand quickly with a clean set of data. Each team member, including me, got a unique and independent test environment.

Even before the team mastered TDD, the adopted infrastructure was in place to support executing tests. This infrastructure enabled the team to start testing much more effectively. Another aspect of trying to automate testing was dealing with a legacy application that was difficult to test. The decisions that were made to enable TDD also helped with customer-facing tests. We decided to start rewriting the system in a new architecture that facilitated testing and test automation, not only at the unit level but at all levels.

—Lisa

Writing tests and writing code with those tests in mind means programmers are always consciously making code testable. All of these good infrastructure-related qualities spill over to business-facing tests and tests that critique the product. The whole team is continually thinking of ways to improve design and make testing easier.

Designing with Testing in Mind

One advantage of driving development with tests is that code is written with the express intention of making the tests pass. The team has to think, right from the beginning, about how it will execute and automate tests for every story it codes. Test-driven development means that programmers will write each test before they write the code to make it pass.

Writing "testable code" is a simple concept, but it's not an easy task, especially if you're working on old code that has no automated tests and isn't designed for testability. Legacy systems often have business logic, I/O, database, and user interface layers intertwined. There's no easy way to hook in to automate a test below the GUI or at the unit level.

A common approach in designing a testable architecture is to separate the different layers that perform different functions in the application. Ideally, you would want to access each layer directly with a test fixture and test algorithms with different inputs. To do this, you isolate the business logic into its own layer, using fake objects instead of trying to access other applications or the actual database. If the presentation layer can be separated from underlying business logic and database access, you can quickly test input validation without testing underlying logic.

Layered Architectures and Testability

Lisa's team took the "strangler application" approach to creating a testable system where tests could be use to drive coding. Mike Thomas, the team's senior architect, explains how their new layered architecture enabled a testable design.

> A layered architecture divides a code base into horizontal slices that contain similar functionality, often related to a technology. The slices at the highest level are the most specific and depend upon the slices below, which are more general. For example, many layered code bases have slices such as the following: UI, business logic, and data access.

Horizontal layering is just one way to organize a code base: Another is domain-oriented slices (such as payroll or order entry), which are generally thought of as "vertical." These layering approaches can be combined, of course, and all can be used to enhance testability.

Layering has advantages for testing, but only if the mechanism for "connecting" the slices provides flexibility. If a code base has tightly coupled slices via such mechanisms as direct concrete class dependencies and static methods, it is difficult to isolate a unit for testing, despite the layering. This makes most automated tests into integration tests, which can be complicated and can run slowly. In many cases, testing can only be accomplished by running the entire system.

Contrast this with a code base where the layers are separated by interfaces. Each slice depends only upon interfaces defined in the slice beneath it rather than on specific classes. Dependencies on such interfaces are easy to satisfy with test doubles at test time: mocks, stubs, and so on. Unit testing is thus simplified because each unit can truly be isolated. For example, the UI can be tested against mock business layer objects, and the business layer can be tested against mock data access objects, avoiding live database access.

The layered approach has allowed Lisa's team to succeed in automating tests at all levels and drive development with both technology-facing and business-facing tests.

See the bibliography for more information on Alastair Cockburn's Ports and Adapters pattern.

Another example of an approach to testable design is Alistair Cockburn's Ports and Adapters pattern [Cockburn, 2005]. This pattern's intent is to "create your application to work without either a UI or a database so you can run automated regression tests against the application, work when the database becomes unavailable, and link applications together without any user involvement." Ports accept outside events, and a technology-specific adapter converts it into a message that can be understood by the application. In turn, the application sends output via a port to an adapter, which creates the signals needed by the receiving human or automated users. Applications designed using this pattern can be driven by automated test scripts as easily as by actual users.

It's more obvious how to code test-first on a greenfield project. Legacy systems, which aren't covered by automated unit tests, present a huge challenge. It's hard to write unit tests for code that isn't designed for testability, and it's hard to change code that isn't safeguarded with unit tests. Many teams have

The bibliography has links to more articles about "rescue" and "strangler" approaches to legacy code.

followed the "legacy code rescue" techniques explained by Michael Feathers in *Working Effectively with Legacy Code* [Feathers, 2005]. Other teams, such as Lisa's, aim to "strangle" their legacy code. This strategy stems from Martin Fowler's "strangler application" [Fowler, 2004]. New stories were coded test-first in a new architecture while the old system was still maintained. Over time, much of the system has been converted to the new architecture, with the goal of eventually doing away with the old system.

Agile testing in a legacy mainframe type of environment presents particular challenges, not the least of which is the lack of availability of publications and information about how to do it successfully. COBOL, mainframes, and their ilk are still widely used. Let agile principles and values guide your team as you look for ways to enable automated testing in your application. You might have to adapt some techniques; for example, maybe you can't write code test-first, but you can test soon after writing the code. When it's the team's problem to solve, and not just the testers' problem, you'll find a way to write tests.

Testing Legacy Systems

John Voris, a developer with Crown Cork and Seal, works in the RPG language, a cousin of COBOL, which runs on the operating system previously known as AS 400 and now known as System i. John was tasked with merging new code with a vendor code base. He applied tenets of Agile, Lean, and IBM-recommended coding practices to come up with an approach he calls "ADEPT" for "AS400 Displays for External Prototyping and Testing." While he isn't coding test-first, he's testing "Minutes Afterward." Here's how he summed up his approach:

For more information about Presenter First development, see the bibliography.

- Write small, single-purpose modules (not monolithic programs), and refactor existing programs into modules. Use a Presenter First development approach (similar to the Model View Presenter or Model View Controller pattern).
- Define parameter interfaces for the testing harness based on screen formats and screen fields. The only drawback here is numbers are defined as zoned decimals rather than packed hexadecimal, but this is offset by the gain in productivity.
- "Minutes after" coding each production module, create a testing program using the screen format to test via the UI. The UI interface for the test is created prior to the production program, because the UI testing interface is the referenced interface for the production module. The impetus for running a test looms large for the programmer, because most of the coding for the test is already done.

For more about RPGUnit, see www.RPGUnit.org.

- Use standard test data sets, which are unchanging, canonical test data, to drive the tests.
- This approach, in which the test programs are almost auto-generated, lends itself to automation with a record/playback tool that would capture data inputs and outputs, with tests run in a continuous build, using RPGUnit.

Your team can find an approach to designing for testability that works for you. The secret is the whole-team commitment to testing and quality. When a team is constantly working to write tests and make them pass, it finds a way to get it done. Teams should take time to consider how they can create an architecture that will make automated tests easy to create, inexpensive to maintain, and long-lived. Don't be afraid to revisit the architecture if automated tests don't return enough value for the investment in them.

Timely Feedback

The biggest value of unit tests is in the speed of their feedback. In our opinion, a continuous integration and build process that runs the unit tests should finish within ten minutes. If each programmer checks code in several times a day, a longer build and test process will cause changes to start stacking up. As a tester, it can be frustrating to have to wait a long time for new functionality or a bug fix. If there's a compile error or unit test failure, the delay gets even worse, especially if it's almost time to go home!

A build and test process that runs tests above the unit level, such as functional API tests or GUI tests, is going to take longer. Have at least one build process that runs quickly, and a second that runs the slower tests. There should be at least one daily "build" that runs all of the slower functional tests. However, even that can be unwieldy. When a test fails and the problem is fixed, how long will it take to know for sure that the build passes again?

If your build and test process takes too long, ask your team to analyze the cause of the slowdown and take steps to speed up the build. Here are a few examples.

- Database access usually consumes lots of time, so consider using fake objects, where possible, to replace the database, especially at the unit level.
- Move longer-running integration and database-access tests to the secondary build and test process.
- See if tests can run in parallel so that they finish faster.

- Run the minimum tests needed for regression testing your system.
- Distribute tasks across multiple build machines.
- Upgrade the hardware and software that run the build.
- Find the area that takes the most time and take incremental steps to speed it up.

Lisa's Story

Early in my current team's agile evolution, we had few unit tests, so we included a few GUI smoke tests in our continual build, which kicked off on every check-in to the source code control system. When we had enough unit tests to feel good about knowing when code was broken, we moved the GUI tests and the FitNesse functional tests into a separate build and test process that ran at night, on the same machine as our continual build.

Our continual ongoing build started out taking less than 10 minutes, but soon was taking more than 15 minutes to complete. We wrote task cards to diagnose and fix the problem. The unit tests that the programmers had written early on weren't well designed, because nobody was sure of the best way to write unit tests. Time was budgeted to refactor the unit tests, use mock data access objects instead of the real database, and redesign tests for speed. This got the build to around eight minutes. Every time it has started to creep up, we've addressed the problem with refactoring, removing unnecessary tests, upgrading the hardware, and choosing different software that helped the build run faster.

As our functional tests covered more code, the nightly build broke more often. Because the nightly build ran on the same machine as the continual ongoing one, the only way to verify that the build was "green" again was to stop the ongoing build, which removed our fast feedback. This started to waste everyone's time. We bought and set up another build machine for the longer build, which now also runs continuously. This was much less expensive than spending so much time keeping two builds running on the same machine, and now we get quick feedback from our functional tests as well.

—Lisa

Wow, multiple continuous build and test processes providing constant feedback—it sounds like a dream to a lot of testers. Regression bugs will be caught early, when they're cheapest to fix. This is a great reason for writing technology-facing tests. Can we get too carried away with them, though? Let's look at the line between technology-facing tests and business-facing tests.

WHERE DO TECHNOLOGY-FACING TESTS STOP?

We often hear people worry that the customer-facing tests will overlap so much with the technology-facing tests that the team will waste time. We know that

business-facing tests might cover a bit of the same ground as unit or code integration tests, but they have such different purposes that waste isn't a worry.

For example, we have a story to calculate a loan amortization schedule and display it to a user who's in the process of requesting a loan. A unit test for this story would likely test for illegal arguments, such as an annual payment frequency if the business doesn't allow it. There might be a unit test to figure the anticipated loan payment start date given some definition of amount, interest rate, start date, and frequency. Unit-level tests could cover different combinations of payment frequency, amount, interest date, term, and start date in order to prove that the amortization calculation is correct. They could cover scenarios such as leap years. When these tests pass, the programmer feels confident about the code.

Each unit test is independent and tests one dimension at a time. This means that when a unit test fails, the programmer can identify the problem quickly and solve the issue just as quickly. The business-facing tests very seldom cover only one dimension, because they are tackled from a business point of view.

The business-facing tests for this story would define more details for the business rules, the presentation in the user interface, and error handling. They would verify that payment details, such as the principal and interest applied, display correctly in the user interface. They would test validations for each field on the user interface, and specify error handling for situations such as insufficient balance or ineligibility. They could test a scenario where an administrator processes two loan payments on the same day, which might be harder to simulate at the unit level.

The business-facing tests cover more complex user scenarios and verify that the end user will have a good experience. Push tests to lower levels whenever possible; if you identify a test case that can be automated at the unit level, that's almost always a better return on investment.

Chapter 13, "Why We Want to Automate Tests and What Holds Us Back," talks more about the ROI of the different types of tests.

If multiple areas or layers of the application are involved, it might not be possible to automate at the unit level. Both technology-facing and business-facing levels might have tests around the date of the first loan payment, but they check for different reasons. The unit test would check the calculation of the date, and the business-facing test would verify that it displays correctly in the borrower's loan report.

Learning to write Quadrant 1 tests is hard. Many teams making the transition to agile development start out with no automated unit tests, not even a

continuous integration and build process. In the next section, we suggest actions agile testers can take if their teams don't tackle Quadrant 1 tests.

WHAT IF THE TEAM DOESN'T DO THESE TESTS?

Many an organization has decided to try agile development, or at least stated that intention, without understanding how to make a successful transition. When we're in a tester role, what can we do to help the development team implement TDD, continuous integration, and other practices that are key to successful development?

Our experience over the years has been that if we aren't programmers ourselves, we don't necessarily have much credibility when we urge the programmers to adopt practices such as TDD. If we could sit down and show them how to code test-first, that would be persuasive, but many of us testers don't have that kind of experience. We've also found that evangelizing doesn't work. It's not that hard to convince someone conceptually that TDD is a good idea. It's much trickier to help them get traction actually coding test-first.

What Can Testers Do?

If you're a tester on a so-called "agile" team that isn't even automating unit tests or producing continuous builds—or at a minimum, doing builds on a daily basis—you're going to get frustrated pretty quickly. Don't give up; keep brainstorming for a way to get traction on a positive transition. Try using social time or other relaxing activity to take some quality time to see what new ideas you can generate to get all team members on board.

One trap to avoid is having testers write the unit tests. Because TDD is really more of a design activity, it's essential that the person writing the code also write the tests, before writing the code. Programmers also need the immediate feedback that automated unit tests give. Unit tests written by someone else after the code is written might still guard against regression defects, but they won't have the most valuable benefits of tests written by the programmer.

Lisa's Story

Whenever I've wanted to effect change, I've turned to the patterns in *Fearless Change* by Mary Lynn Manns and Linda Rising [2004]. After working on two XP teams, I joined a team that professed a desire to become agile but wasn't making strides toward solid development practices. I found several patterns in *Fearless Change* to try to move the team toward agile practices.

"Ask for Help" was one pattern that helped me. This pattern says, in part: "Since the task of introducing a new idea into an organization is a big job, look for people and resources to help your efforts" [Manns and Rising, 2004]. When I wanted my team to start using FitNesse, I identified the programmer who was most sympathetic to my cause and asked him to pair with me to write FitNesse tests for the story he was working on. He told the other programmers about the benefits he derived from the FitNesse tests, which encouraged them to try it too. Most people want to help, and agile is all about the team working together, so there's no reason to go it alone.

"Brown Bag" is another change pattern that my teams have put to good use. For example, my current team held several brown bag sessions where they wrote unit tests together. "Guru on Your Side" is a productive pattern in which you enlist the help of a well-respected team member who might understand what you're trying to achieve. A previous team I was on was not motivated to write unit tests. The most experienced programmer on the team agreed with me that test-driven development was a good idea, and he set an example for the rest of the team.

We think you'll find that there's always someone on an agile team who's sympathetic to your cause. Enlist that person's support, especially if the team perceives him or her as a senior-level guru.

—Lisa

As a tester on an agile team, there's a lot you can do to act as a change agent, but your potential impact is limited. In some cases, strong management support is the key to driving the team to engage in Quadrant 1 activities.

What Can Managers Do?

If you're managing a development team, you can do a lot to encourage test-driven development and unit test automation. Work with the product owner to make quality your goal, and communicate the quality criteria to the team. Encourage the programmers to take time to do their best work instead of worrying about meeting a deadline. If a delivery date is in jeopardy, push to reduce the scope, not the quality. Your job is to explain to the business managers how making quality a priority will ensure that they get optimum business value.

Give the team time to learn, and provide expert, hands-on training. Bring in an experienced agile development coach or hire someone with experience in using these practices who can transfer those skills to the rest of the team. Budget time for major refactoring, for brainstorming about the best approach to writing unit and code integration tests, and for evaluating, installing, and upgrading tools. Test managers should work with development

managers to encourage practices that enhance testability and allow testers to write executable tests. Test managers can also make sure testers have time to learn how to use the automation tools and frameworks that the team decides to implement.

It's a Team Problem

More about retrospectives and process improvement in Chapter 19, "Wrap Up the Iteration."

While you can find ways to be an effective change agent, the best thing to do is involve the whole team in solving the problems. If you aren't already doing retrospectives after every iteration, propose trying this practice or some other type of process improvement. At the retrospective, raise issues that are hampering successful delivery. For example, "We aren't finishing testing tasks before the end of the iteration" is a problem for the whole team to address. If one reason for not finishing is the high number of unit-level bugs, suggest experimenting with TDD, but allow programmers to propose their own ways to address the problem. Encourage the team to try a new approach for a few iterations and see how it works.

Technology-facing tests that support the team's development process are an important foundation for all of the testing that needs to happen. If the team isn't doing an adequate job with the tests in this quadrant, the other types of testing will be much more difficult. This doesn't mean you can't get value from the other quadrants on their own—it just means it will be harder to do so because the team's code will lack internal quality and everything will take longer.

Technology-facing tests can't be done without the right tools and infrastructure. In the next section, we look at examples of the types of tools a team needs to be effective with Quadrant 1 tests.

TOOLKIT

There's no magical tool that will ensure success. However, tools can help good people do their best work. Building up the right infrastructure to support technology-facing tests is critical. There's a huge selection of excellent tools available, and they improve all the time. Your team must find the tools that work best for your situation.

Source Code Control

Source code control is known by other names too, such as version control or revision control. It's certainly not new, or unique to agile development, but

no software development team can succeed without it. That's why we're discussing it here. Without source code control, you'll never be sure what you're testing. Did the programmer change only the module he said he changed, or did he forget changes he made to other modules? You can't back out unwanted or erroneous changes without some kind of versioning system. Source code control keeps different programmers from walking on each other's changes to the same modules. Without versioning, you can't be sure what code to release to production.

Software Configuration Management Patterns: Effective Teamwork, Practical Integrations [2003], by Stephen Berczuk and Brad Appleton, is a good resource to use to learn how and why to use source code control. Source code control is essential to any style of software development.

Use source code control for automated test scripts, too. It's important to tie the automated tests with the corresponding code version that they tested in case you need to rerun tests against that version in the future. When you label or tag a build, make sure you label or tag the test code too, even if it doesn't get released to production.

Teams can organize their code hierarchy to provide a repository for production code, corresponding unit tests, and higher-level test scripts. Doing this might require some brainstorming and experimenting in order to get the right structure.

There are many terrific options to choose from. Open source systems such as CVS and Subversion (SVN) are easy to implement, integrate with a continuous build process and IDEs, and are robust. Vendor tools such as IBM Rational ClearCase and Perforce might add features that compensate for the increased overhead they often bring.

Source code control is tightly integrated with development environments. Let's look at some IDEs used by agile teams.

IDEs

A good IDE (integrated development environment) can be helpful for programmers and testers on an agile team. The IDE integrates with the source code control system to help prevent problems with versioning and changes walking on each other. The editors inside an IDE are specific to the programming language and flag errors even as you write the code. Most importantly, IDEs provide support for refactoring.

Programmers who use an IDE tend to have strong personal preferences. However, sometimes an organization decrees that all programmers must use a specific IDE. This might be because of licensing, or it might be intended to encourage open pair programming. It is easier to pair with another programmer if the other person uses the same IDE, but it's generally not essential for the same one to be used. Most tools work similarly, so it's not hard to change from one IDE to another in order to meet new needs or take advantage of new features. Some diehards still prefer to use tried-and-true technology such as vi, vim, or emacs with make files rather than an IDE.

Open source IDEs such as Eclipse and NetBeans are widely used by agile teams, along with proprietary systems such as Visual Studio and IntelliJ IDEA. IDEs have plug-ins to support different languages and tools. They work as well with test scripts as they do with production code.

Lisa's Story

On my current team, some programmers were using IntelliJ IDEA, while others used Eclipse. Environmental differences in rare cases caused issues, such as tests passing in the IDE but not the full build, or check-ins via the IDE causing havoc in the source code control system. Generally, though, use of different IDEs caused no problems. Interestingly, over time most of the Eclipse users switched. Pairing with the IntelliJ users led them to prefer it.

I use Eclipse to work with the automated test scripts as well as to research issues with the production code. The Ruby plug-in helps us with our Ruby and Watir scripts, and the XML editor helps with our Canoo WebTest scripts. We can run unit tests and do builds through the IDE. Programmers on the team helped me set up and start using Eclipse, and it has saved huge amounts of time. Maintaining the automated tests is much easier, and the IDE's "synchronize" view helps me remember to check in all of the modules I've changed.

Test tools are starting to come out with their own IDEs or plug-ins to work with existing IDEs such as Eclipse. Take advantage of these powerful, time-saving, quality-promoting tools.

—Lisa

Testers who aren't automating tests through an IDE, but who want to be able to look at changed snippets of code, can use tools such as FishEye that enable the testers to get access to the code through the automated build.

As of this writing, IDEs have added support for dynamic languages such as Ruby, Groovy, and Python. Programmers who use dynamic languages may prefer lighter-weight tools, but they still need good tools that support good coding practices, such as TDD and refactoring.

Regardless of the development environment and tools being used, agile teams need a framework that will integrate code changes from different programmers, run the unit tests to verify no regression bugs have occurred, and provide the code in a deployable format.

Build Tools

Your team needs some way to build the software and create a deployable jar, war, or other type of file. This can be done with shell-based tools such as make, but those tools have limitations, such as the platforms where they work. Agile teams that we know use tools such as ant, Nant, and Maven to build their projects. These tools not only manage the build but also provide easy ways to report and document build results, and they integrate easily with build automation and test tools. They also integrate with IDEs.

Build Automation Tools

Continuous integration is a core practice for agile teams. You need a way to not only build the project but also run automated tests on each build to make sure nothing broke. A fully automated and reproducible build that runs many times a day is a key success factor for agile teams. Automated build tools provide features such as email notification of build results, and they integrate with build and source code control tools.

Commonly used tools as of the writing of this book include the open source tools CruiseControl, CruiseControl.net, CruiseControl.rb, and Hudson. Other open source and proprietary tools available at publication time are AnthillPro, Bamboo, BuildBeat, CI Factory, Team City, and Pulse, just to name a few.

Without an automated build process you'll have a hard time deploying code for testing as well as releasing. Build management and build automation tools are easy to implement and absolutely necessary for successful agile projects. Make sure you get your build process going early, even before you start coding. Experiment with different tools when you find you need more features than your current process provides.

Unit Test Tools

Unit test tools are specific to the language in which you're coding. "xUnit" tools are commonly used by agile teams, and there's a flavor for many different languages, including JUnit for Java, NUnit for .NET, Test::Unit for Perl and Ruby, and PyUnit for Python.

Behavior-driven development is another flavor of test-driven development, spelling out expected behavior to drive tests with tools such as RSpec and easyb.

See Chapter 9, "Toolkit for Business-Facing Tests that Support the Team," for more information on behavior-driven development tools.

GUI code can and should be developed test-first as well. Some tools for rich-client unit testing are TestNG, Abbot, and SWTBot.

Tools such as EasyMock and Ruby/Mock help with implementing mock objects and test stubs, an integral part of well-designed unit tests.

See the bibliography for links and books to help your team search for the right unit test tools.

The tools programmers use to write technology-facing tests can also be used for business-facing tests. Whether they are suited for that purpose in your project depends on the needs of your team and your customers.

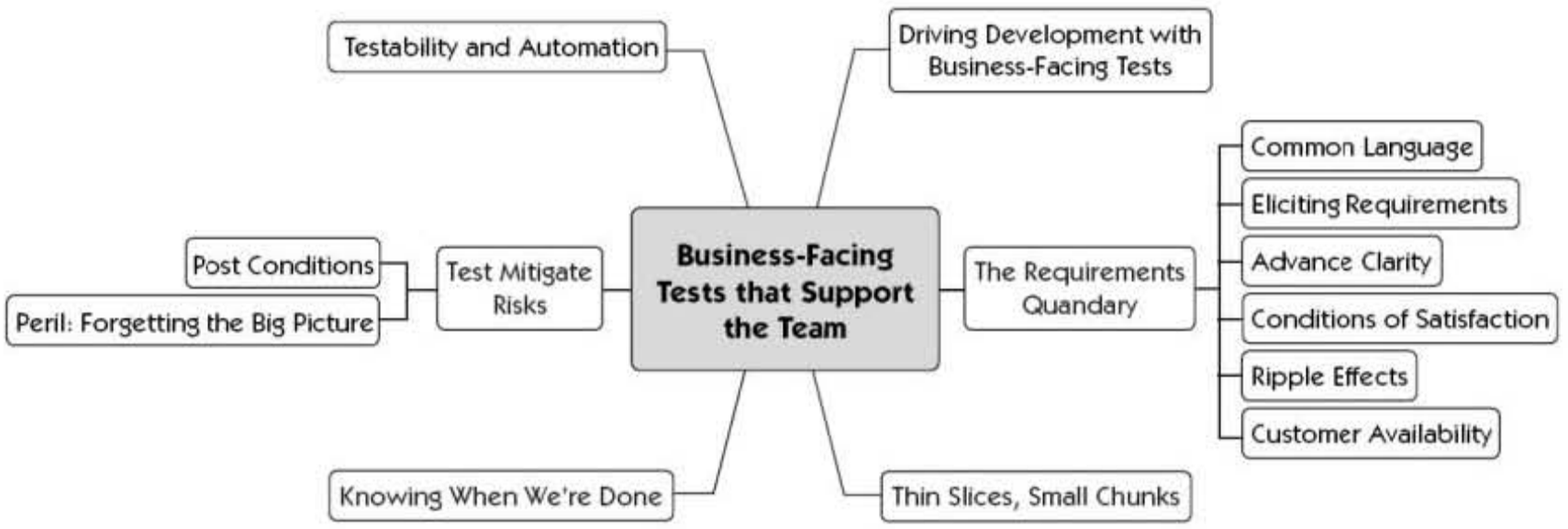
SUMMARY

In this chapter, we explained the purpose of technology-facing tests that support the team, and we talked about what teams need to use them effectively.

- Technology-facing tests that support programming let the team produce the highest quality code possible; they form the foundation for all other types of testing.
- The benefits of this quadrant's tests include going faster and doing more, but speed and quantity should never be the ultimate goal.
- Programmers write technology-facing tests that support the team and provide great value to testers by enhancing the internal quality and testability of the system.
- Teams that fail to implement the core practices related to agile development are likely to struggle.
- Legacy systems usually present the biggest obstacles to test-driven development, but these problems can be overcome with incremental approaches.
- If your team doesn't now do these tests, you can help them get started by engaging other team members and getting support from management.
- There can be some overlap between technology-facing tests and business-facing tests that support the team. However, when faced with a choice, push tests to the lowest level in order to maximize ROI.
- Teams should set up continuous integration, build, and test processes in order to provide feedback as quickly as possible.
- Agile teams require tools for tasks such as source code control, test automation, IDEs, and build management in order to facilitate technology-facing tests that support the team.

Chapter 8

Business-Facing Tests that Support the Team



In the last chapter, we talked about programmer tests, those low-level tests that help programmers make sure they have written the code right. How do they know the right thing to build? In phased and gated methodologies, we try to solve that by gathering requirements up front and putting as much detail in them as possible. In projects using agile practices, we put all our faith in story cards and tests that customers understand in order to help code the right thing. These "understandable" tests are the subject of this chapter.

Driving Development with Business-Facing Tests

Yikes, we're starting an iteration with no more information than what fits on an index card, something like what's shown in Figure 8-1.

That's not much information, and it's not meant to be. Stories are a brief description of desired functionality and an aid to planning and prioritizing work. On a traditional waterfall project, the development team might be

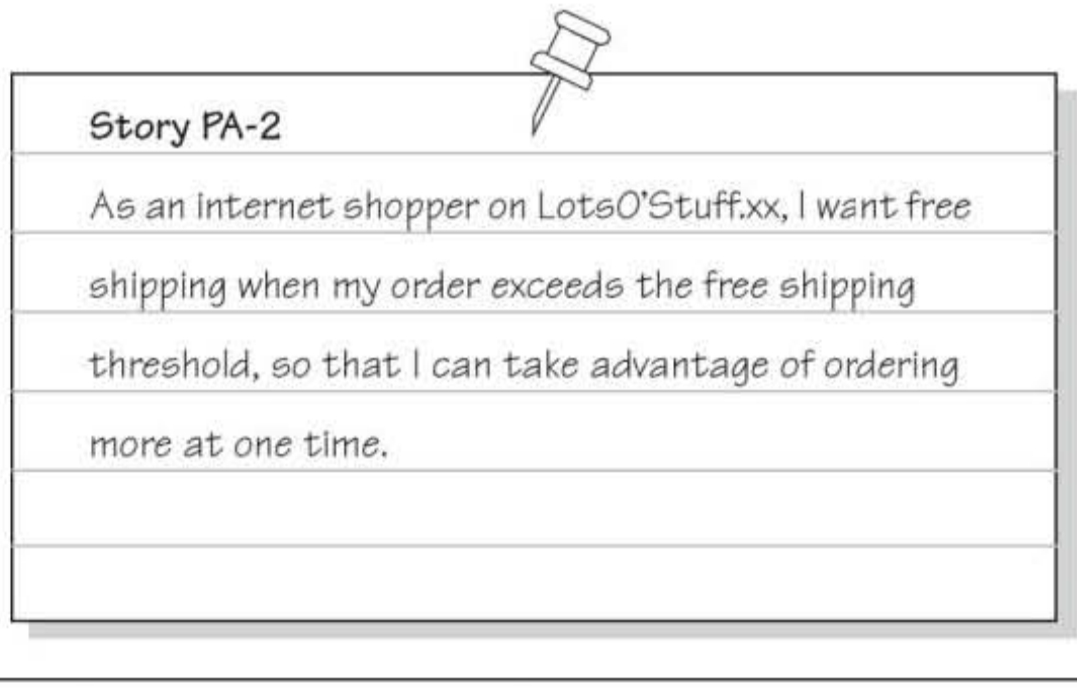


Figure 8-1 Story to set up conversation

given a wordy requirements document that includes every detail of the feature set. On an agile project, the customer team and development team strike up a conversation based on the story. The team needs requirements of some kind, and they need them at a level that will let them start writing working code almost immediately. To do this, we need examples to turn into tests that will confirm what the customer really wants.

These business-facing tests address business requirements. These tests help provide the big picture and enough details to guide coding. Business-facing tests express requirements based on examples and use a language and format that both the customer and development teams can understand. Examples form the basis of learning the desired behavior of each feature, and we use those examples as the basis for our story tests in Quadrants 2 (see Figure 8-2).

Business-facing tests are also called "customer-facing," "story," "customer," and "acceptance" tests. The term "acceptance test" is particularly confusing, because it makes some people think only of "user acceptance tests." In the context of agile development, acceptance tests generally refer to the business-facing tests, but the term could also include the technology-facing tests from Quadrant 4, such as the customer's criteria for system performance or security. In this chapter, we're discussing only the business-facing tests that support the team by guiding development and providing quick feedback.

As we explained in the previous two chapters, the order in which we present these four quadrants isn't related to the order in which we might perform

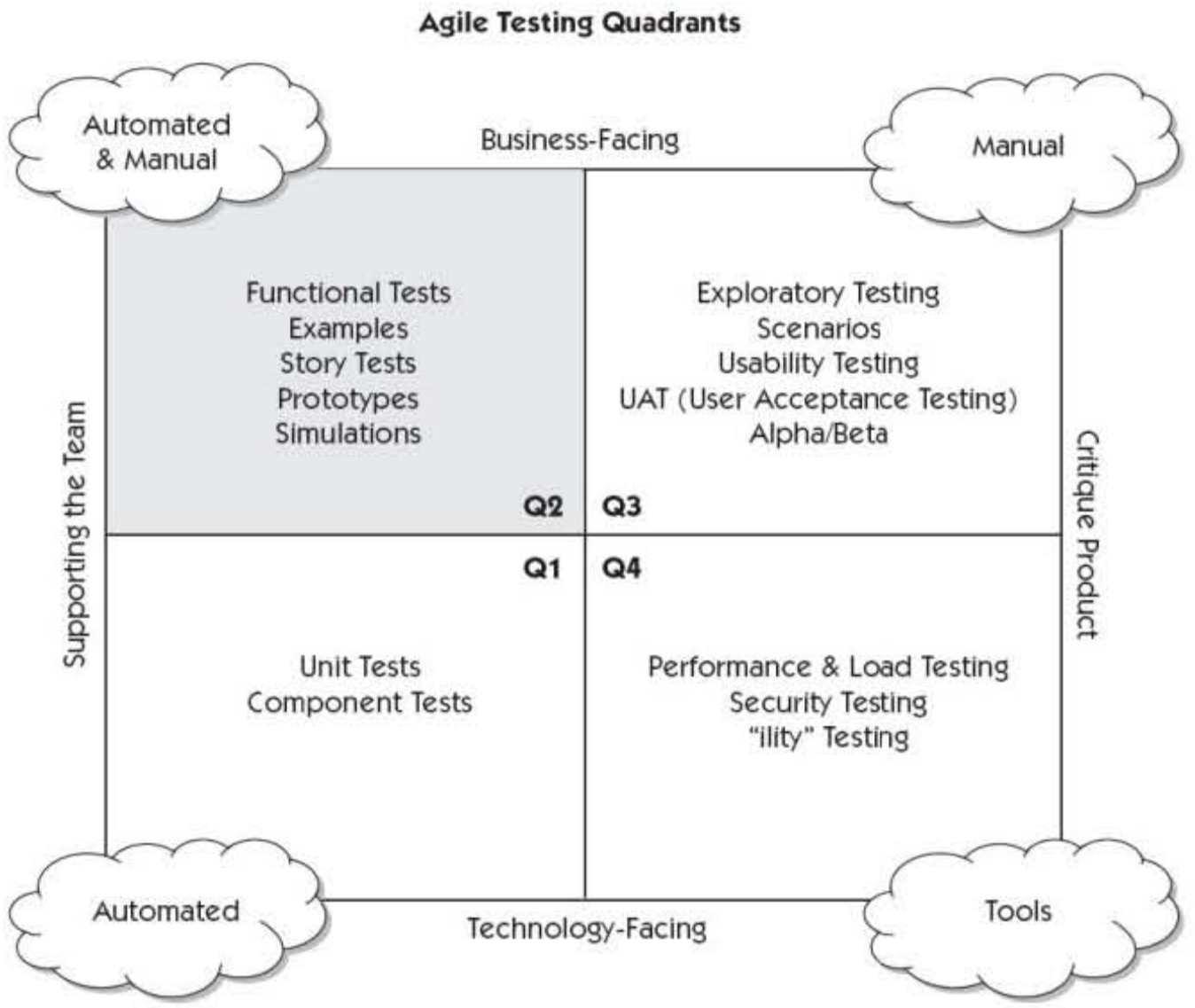


Figure 8-2 The Agile Testing Quadrants, highlighting Quadrant 2

Part V, "An Iteration in the Life," examines the order in which we perform tests from the different quadrants.

activities from each quadrant. The business-facing tests in Quadrant 2 are written for each story before coding is started, because they help the team understand what code to write. Like the tests in Quadrant 1, these tests drive development, but at a higher level. Quadrant 1 activities ensure internal quality, maximize team productivity, and minimize technical debt. Quadrant 2 tests define and verify external quality, and help us know when we're done.

The customer tests to drive coding are generally written in an executable format, and automated, so that team members can run the tests as often as they like in order to see if the functionality works as desired. These tests, or some subset of them, will become part of an automated regression suite so that future development doesn't unintentionally change system behavior.

As we discuss the stories and examples of desired behavior, we must also define nonfunctional requirements such as performance, security, and usability. We'll

also make note of scenarios for manual exploratory testing. We'll talk about these other types of testing activities in the chapters on Quadrants 3 and 4.

We hear lots of questions relating to how agile teams get requirements. How do we know what the code we write should do? How do we obtain enough information to start coding? How do we get the customers to speak with one voice and present their needs clearly? Where do we start on each story? How do we get customers to give us examples? How do we use those to write story tests?

This chapter explains our strategy for creating business-facing tests that support the team as it develops each story. Let's start by talking more about requirements.

THE REQUIREMENTS QUANDARY

Just about every development team we've known, agile or not, struggles with requirements. Teams on traditional waterfall projects might invest months in requirements gathering only to have them be wrong or quickly get out of date. Teams in chaos mode might have no requirements at all, with the programmers making their best guess as to how a feature should work.

Agile development embraces change, but what happens when requirements change during an iteration? We don't want a long requirements-gathering period before we start coding, but how can we be sure we (and our customers) really understand the details of each story?

In agile development, new features usually start out life as stories, or groups of stories, written by the customer team. Story writing is not about figuring out implementation details, although high-level discussions can have an impact on dependencies and how many stories are created. It's helpful if some members of the technical team can participate in story-writing sessions so that they can have input into the functionality stories and help ensure that technical stories are included as part of the backlog. Programmers and testers can also help customers break stories down to appropriate sizes, suggest alternatives that might be more practical to implement, and discuss dependencies between stories.

Stories by themselves don't give much detail about the desired functionality. They're usually just a sentence that expresses who wants the feature, what the feature is, and why they want it. "As an Internet shopper, I need a way to delete items from my shopping cart so I don't have to buy unwanted items" leaves a

lot to the imagination. Stories are only intended as a starting point for an ongoing dialogue between business experts and the development team. If team members understand what problem the customer is trying to solve, they can suggest alternatives that might be simpler to use and implement.

In this dialogue between customers and developers, agile teams expand on stories until they have enough information to write appropriate code. Testers help elicit examples and context for each story, and help customers write story tests. These tests guide programmers as they write the code and help the team know when it has met the customers' conditions of satisfaction. If your team has use cases, they can help to supplement the example or coaching test to clarify the needed functionality (see Figure 8-3).

In agile development, we accept that we'll never understand all of the requirements for a story ahead of time. After the code that makes the story tests pass is completed, we still need to do more testing to better understand the requirements and how the features should work.

After customers have a chance to see what the team is delivering, they might have different ideas about how they want it to work. Often customers have a vague idea of what they want and a hard time defining exactly what that is. The team works with the customer or customer proxy for an iteration and might deliver just a kernel of a solution. The team keeps refining the functionality over multiple iterations until it has defined and delivered the feature.

Being able to iterate is one reason agile development advocates small releases and developing one small chunk at a time. If our customer is unhappy with the behavior of the code we deliver in this iteration, we can quickly rectify that in the next, if they deem it important. Requirements changes are pretty much inevitable.

We must learn as much as we can about our customers' wants and needs. If our end users work in our location, or it's feasible to travel to theirs, we should sit with them, work alongside them, and be able to do their jobs if we can. Not only will we understand their requirements better but we might even identify requirements they didn't think to state.

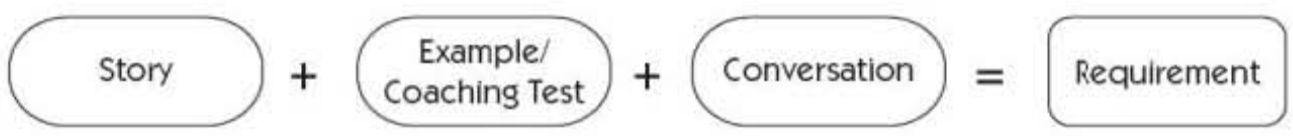


Figure 8-3 The makeup of a requirement

Tests need to include more than the customers' stated requirements. We need to test for post conditions, impact on the system as a whole, and integration with other systems. We identify risks and mitigate those with tests as needed. All of these factors guide our coding.

Common Language

We can also use our tests to provide a common language that's understood by both the development team and the business experts. As Brian Marick [2004] points out, a shared language helps the business people envision the features they want. It helps the programmers craft well-designed code that's easy to extend. Real-life examples of desired and undesired behavior can be expressed so that they're understood by both the business and technical sides. Pictures, flow diagrams, spreadsheets, and prototypes are accessible to people with different backgrounds and viewpoints. We can use these tools to find examples and then easily turn those examples into tests. The tests need to be written in a way that's comprehensible to a business user reading them yet still executable by the technical team.

More on Fit in Chapter 9, "Toolkit for Business-Facing Tests that Support the Team."

Business-facing tests also help define scope, so that everyone knows what is part of the story and what isn't. Many of the test frameworks now allow teams to create a domain language and define tests using that language. Fit (Functional for Integrated Framework) is one of those.

The Perfect Customer

Andy Pols allowed us to reprint this story from his blog [Pols, 2008]. In it, he shows how his customer demanded a test, wrote it, and realized the story was out of scope.

On a recent project, our customer got so enthusiastic about our Fit tests that he got extremely upset when I implemented a story without a Fit test. He refused to let the system go live until we had the Fit test in place.

The story in question was very technical and involved sending a particular XML message to an external system. We just could not work out what a Fit test would look like for this type of requirement. Placing the expected XML message, with all its gory detail, in the Fit test would not have been helpful because this is a technical artifact and of no interest to the business. We could not work out what to do. The customer was not around to discuss this, so I just went ahead and implemented the story (very naughty!).

What the customer wanted was to be sure that we were sending the correct product information in the XML message. To resolve the issue, I suggested that we have a Fit test that shows how the product attributes get mapped onto the XML message using Xpath, although I still thought this was too technical for a business user.

We gave the customer a couple of links to explain what XPath was so that he could explore whether this was a good solution for him. To my amazement, he was delighted with XPath (I now know who to turn to when I have a problem with XPath) and filled in the Fit test.

The interesting bit for me is that as soon as he knew what the message looked like and how it was structured, he realized that it did not really support the business—we were sending information that was outside our scope of our work and that should have been supplied by another system. He was also skeptical about the speed at which the external team could add new products due to the complex nature of the XML.

Most agile people we tell this story to think we have the "perfect customer!"

Even if your customers aren't perfect, involving them in writing customer tests gives them a chance to identify functionality that's outside the scope of the story. We try to write customer tests that customers can read and comprehend. Sometimes we set the bar too low. Collaborate with your customers to find a tool and format for writing tests that works for both the customer and development teams.

It's fine to say that our customers will provide to us the examples that we need to have in order for us to understand the value that each story should deliver. But what if they don't know how to explain what they want? In the next section, we'll suggest ways to help customers define their conditions of satisfaction.

Eliciting Requirements

If you've ever been a customer requesting a particular software feature, you know how hard it is to articulate exactly what you want. Often, you don't really know exactly what you want until you can see, feel, touch and use it. We have lots of ways to help our customers get clarity about what they want.

Ask Questions

Start by asking questions. Testers can be especially good at asking a variety of questions because they are conscious of the big picture, the business-facing

and technical aspects of the story, and are always thinking of the end user experience. Types of general questions to ask are:

- Is this story solving a problem?
- If so, what's the problem we're trying to solve?
- Could we implement a solution that doesn't solve the problem?
- How will the story bring value to the business?
- Who are the end users of the feature?
- What value will they get out of it?
- What will users do right before and right after they use that feature?
- How do we know we're done with this story?

One question Lisa likes to ask is, "What's the worst thing that could happen?" Worst-case scenarios tend to generate ideas. They also help us consider risk and focus our tests on critical areas. Another good question is, "What's the best thing that could happen?" This question usually generates our happy path test, but it might also uncover some hidden assumptions.

Use Examples

Most importantly, ask the customer to give you examples of how the feature should work. Let's say the story is about deleting items out of an online shopping cart. Ask the customer to draw a picture on a whiteboard of how that delete function might look. Do they want any extra features, such as a confirmation step, or a chance to save the item in case they want to retrieve it later? What would they expect to see if the deletion couldn't be done?

Examples can form the basis for our tests. Our challenge is to capture examples, which might be expressed in the business domain language, as tests that can actually be executed. Some customers are comfortable expressing examples using a test tool such as Fit or FitNesse as long as they can write them in their domain language.

Let's explore the difference between an example and a test with a simple story (see Figure 8-4). People often get confused between these two terms.

An example would look something like this:

> *There are 5 items on a page. I want to select item 1 for $20.25 and put it in the shopping cart. I click to the next page, which has 5 more items. I select a second item on that page for $5.38 and put it in my shopping cart. When I say I'm done shopping, it will show both the item from the*

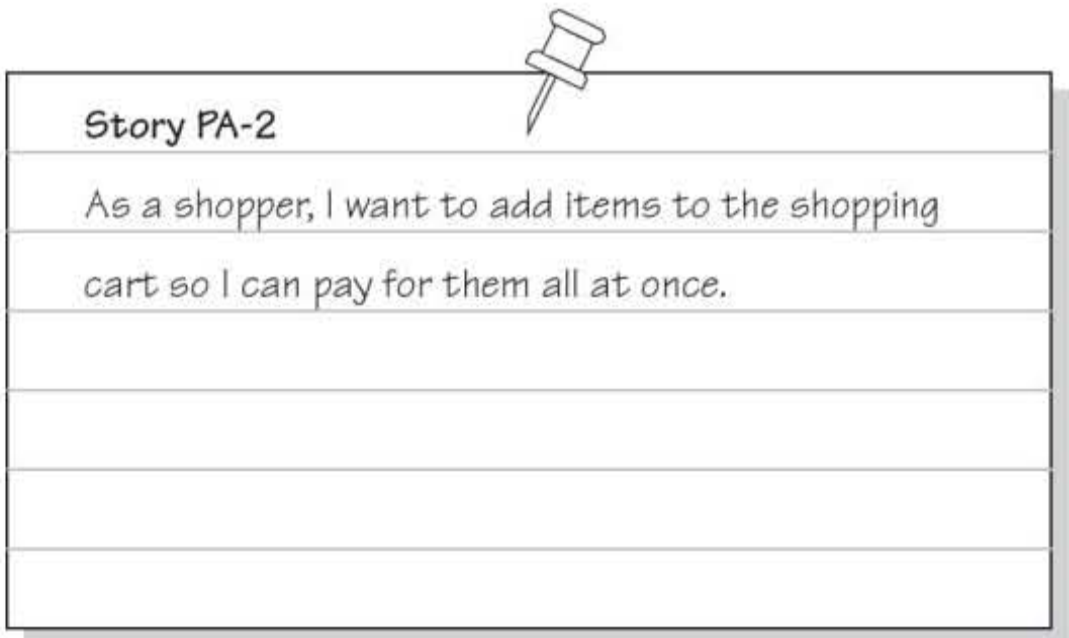


Figure 8-4 Story to use as a base for examples and tests

first page and the item from the second page in my shopping cart, with the total of $25.63

The test could be quite a bit different. We'll use a Fit type format in Table 8-1 to show you how the test could be represented.

The test captures the example in an executable format. It might not use exactly the same inputs, but it encapsulates the sample user scenario. More test cases can be written to test boundary conditions, edge cases, and other scenarios.

Multiple Viewpoints

Each example or test has one point of view. Different people will write different tests or examples from their unique perspectives. We'd like to capture as many different viewpoints as we can, so think about your users.

Table 8-1 Test for Story PA-2

	Inputs		Expected Results	
ID	**Item**	**Price**	**Total Cost**	**# of Items**
001	Item A	20.25	20.25	1
002	Item D	0.01	20.26	2
003	Item F	100.99	121.25	3

Getting the requirements right is an area where team members in many different roles can jump in to help. Business analysts, subject matter experts, programmers, and various members of the customer team all have something to contribute. Think about other stakeholders, such as your production support team. They have a very unique perspective.

We often forget about nonfunctional requirements such as "How long does the system need to be up? What happens if it fails? If we have middleware that passes messages, do we expect messages to be large enough that we might need to consider loss during transmission? Or will they be a constant size? What happens if there is no traffic for hours? Does the system need to warn someone?" Testing for these types of requirements usually falls into quadrants 3 and 4, but we still need to write tests to make sure they get done.

All of the examples that customers give to the team add up quickly. Do we really have to turn all of these into executable tests? Not as long as we have the customers there to tell us if the code is working the way they want. With techniques such as paper prototyping, designs can be tested before a line of code is written.

Wizard of Oz Testing

Gerard Meszaros, a Certified ScrumMaster (Practicing) and Agile Coach, shared his story about Wizard of Oz Testing on Agile Projects. He describes a good example of how artifacts we generate to elicit requirements can help communicate meaning in an unambiguous form.

> We thought we were ready to release our software. We had been building it one iteration at a time under the guidance of an on-site customer who had prioritized the functionality based on what he needed to enter into integration testing with his business partners. We consciously deferred the master data maintenance and reporting functionality to later iterations to ensure we had the functionality needed for integration testing ready. The integration testing went fine, with just a few defects logged (all related to missing or misunderstood functionality). In the meantime, we implemented the master data maintenance in parallel with integration testing in the last few iterations. When we went into acceptance testing with the business users, we got a rude shock: They hated the maintenance and reporting functionality! They logged so many defects and "must-have improvements" that we had to delay the release by a month. So much for coming up with a plan that would allow us to deliver early!

While we were reimplementing the master data maintenance, I attended the Agile 2005 conference and took a tutorial by Jeff Patton. One of the exercises was building paper prototypes of the UI for a sample application. Then we "tested" the paper prototypes with members of the other groups as our users and found out how badly flawed our UI designs were. Déjà vu! The tutorial resembled my reality.

On my return to the project back home, I took the project manager I was mentoring in agile development aside and suggested that paper prototyping and "Wizard of Oz" testing (the Wizard of Oz reference is to a human being acting as a computer—sort of the "man behind the curtain") might have avoided our one-month setback. After a very short discussion, we decided to give it a try on our release 2 functionality. We stayed late a couple of evenings and designed the UI using screenshots from the R1 functionality overlaid with hand-drawn R2 functionality. It was a long time since either of us had used scissors and glue sticks, and it was fun!

For the Wizard of Oz testing with users, we asked our on-site customers to find some real users with whom to do the testing. They also came up with some realistic sample tasks for the users to try to execute. We put the sample data into Excel spreadsheets and printed out various combinations of data grids to use the in the testing. Some future users came to town for a conference. We hijacked pairs of them for an hour each and did our testing.

I acted as the "wizard," playing the part of the computer ("it's a 286 processor so don't expect the response times to be very good"). The on-site customer introduced the problem and programmers acted as observers, recording the missteps the users made as "possible defects." After just a few hours, we had huge amounts of valuable data about which parts of our UI design worked well and which parts needed rethinking. And there was little argument about which was which! We repeated the usability testing with other users when we had alpha versions of the application available and gained further valuable insights. Our business customer found the exercise so valuable that on a subsequent project the business team set about doing the paper prototyping and Wizard of Oz testing with no prompting from the development team. This might have been influenced somewhat by the first e-mail we got from a real user 30 minutes after going live: "I love this application!!!"

Developing user interfaces test-first can seem like an intimidating effort. The Wizard of Oz technique can be done before writing a single line of code. The team can test user interaction with the system and gather plenty of information to understand the desired system behavior. It's a great way to facilitate communication between the customer and development teams.

Close, constant collaboration between the customer team and the developer team is key to obtaining examples on which to base customer tests that drive coding. Communication is a core agile value, and we talk about it more in the next section.

Communicate with Customers

In an ideal world, our customers are available to us all day, every day. In reality, many teams have limited access to their business experts, and in many cases, the customers are in a different location or time zone. Do whatever you can to have face-to face conversations. When you can't, conference calls, phone conversations, emails, instant messages, cameras, and other communication tools will have to substitute. Fortunately, more tools to facilitate remote communication are available all the time. We've heard of teams, such as Erika Boyer's team at iLevel by Weyerhaeuser, that use webcams that can be controlled by the folks in the remote locations. Get as close to you can to direct conversation.

Lisa's Story

I worked on a team where the programmers were spread through three time zones and the customers were in a different one. We sent different programmers, testers, and analysts to the customer site for every iteration, so that each team member had "face time" with the customers at least every third iteration. This built trust and confidence between the developer and customer teams. The rest of the time we used phone calls, open conference calls, and instant messages to ask questions. With continual fine-tuning based on retrospective discussions, we succeeded in satisfying and even delighting the customers.

—Lisa

Even when customers are available and lines of communication are wide open, communication needs to be managed. We want to talk to each member of the customer team, but they all have different viewpoints. If we get several different versions of how a piece of functionality should work, we won't know what to code. Let's consider ways to get customers to agree on the conditions of satisfaction for each story.

Advance Clarity

If your customer team consists of people from different parts of the organization, there may be conflicting opinions among them about exactly what's intended by a particular story. In Lisa's company, business development wants

features that generate revenue, operations wants features that cut down on phone support calls, and finance wants features that streamline accounting, cash management, and reporting. It's amazing how many unique interpretations of the same story can emerge from people who have differing viewpoints.

Lisa's Story

Although we had a product owner when we first implemented Scrum, we still got different directives from different customers. Management decided to appoint a vice president with extensive domain and operations knowledge as the new product owner. He is charged with getting all of the stakeholders to agree on each story's implications up front. He and the rest of the customer team meet regularly to discuss upcoming themes and stories, and to agree on priorities and conditions of satisfaction. He calls this "advance clarity."

—Lisa

A Product Owner is a role in Scrum. He's responsible not only for achieving advance clarity but also for acting as the "customer representative" in prioritizing stories. There's a downside, though. When you funnel the needs of many different viewpoints through one person, something can be lost. Ideally, the development team should sit together with the customer team and learn how to do the customer's work. If we understand the customer's needs well enough to perform its daily tasks, we have a much better chance of producing software that properly supports those tasks.

Janet's Story

Our team didn't implement the product owner role at first and used the domain experts on the team to determine prioritization and clarity. It worked well, but the achieving consensus took many meetings because each person had different experiences. The product was better for it, but there were trade-offs. The many meetings meant the domain experts were not always available for answering questions from the programmers, so coding was slower than anticipated.

There were four separate project teams working on the same product, but each one was focused on different features. After several retrospectives, and a lot of problem-solving sessions, each project team appointed a Product Owner. The number of meetings was cut down significantly because most business decisions were made by the domain experts on their particular project. Meetings were held for all of the domain experts if there were any differences of opinion, and the Product Owner facilitated bringing consensus on an issue. Decisions were made much faster, the domain experts were more available for answering questions by the team, and were able to keep up with the acceptance tests.

—Janet

However your team chooses to bring together varying viewpoints, it is important that there is only "one voice of the customer" presented to the team.

We said that product owners provide conditions of satisfaction. Let's look more closely at what we mean.

Conditions of Satisfaction

There are conditions of satisfaction for the whole release as well as for each feature or story. Acceptance tests help define the story acceptance. Your development team can't successfully deliver what the business wants unless conditions of satisfaction for a story are agreed to up front. The customer team needs to "speak with one voice." If you're getting different requirements from different stakeholders, you might need to push back and put off the story until you have a firm list of business satisfaction conditions. Ask the customer representative to provide a minimum amount of information on each story so that you can start every iteration with a productive conversation.

The best way to understand the customer team's requirements is to talk with the customers face to face. Because everyone struggles with "requirements," there are tools to help the customer team work through each story. Conditions of satisfaction should include not only the features that the story delivers but also the impacts on the larger system.

Lisa's product owner uses a checklist format to sort out issues such as:

- Business satisfaction conditions
- Impact on existing functions such as the website, documents, invoices, forms, or reports
- Legal considerations
- The impact on regularly scheduled processes
- References to mock-ups for UI stories
- Help text, or who will provide it
- Test cases
- Data migration (as appropriate)
- Internal communication that needs to happen
- External communication to business partners and vendors

Chapter 9, "Toolkit for Business-Facing Tests that Support the Team," includes example checklists as well as other tools for expressing requirements.

The product owner uses a template to put this information on the team's wiki so that it can be used as team members learn about the stories and start writing tests.

These conditions are based on key assumptions and decisions made by the customer team for a story. They generally come out of conversations with the customer about high-level acceptance criteria for each story. Discussing conditions of satisfaction helps identify risky assumptions and increases the team's confidence in writing and correctly estimating all of the tasks that are needed to complete the story.

Ripple Effects

In agile development, we focus on one story at a time. Each story is usually a small component of the overall application, but it might have a big ripple effect. A new story drops like a little stone into the application water, and we might not think about what the resulting waves might run into. It's easy to lose track of the big picture when we're focusing on a small number of stories in each iteration.

Lisa's team finds it helpful to make a list of all of the parts of the system that might be affected by a story. The team can check each "test point" to see what requirements and test cases it might generate. A small and innocent story might have a wide-ranging impact, and each part of the application that it touches might present another level of complexity. You need to be aware of all the potential impacts of any code change. Making a list is a good place to start. In the first few days of the iteration, the team can research and analyze affected areas further and see whether any more task cards are needed to cover them all.

Janet's Story

In one project I was on, we used a simple spreadsheet that listed all of the high-level functionality of the application under test. During release planning, and at the start of each new iteration, we reviewed the list and thought about how the new or changing functionality would affect those areas. That became the starting point for determining what level of testing needed to be done in each functional area. This impact analysis was in addition to the actual story testing and enabled our team to see the big picture and the impact of the changes to the rest of the system.

—Janet

Chapter 16, "Hit the Ground Running," and Chapter 17, "Iteration Kickoff," give examples of when and how teams can plan customer tests and explore the wider impact of each story.

Stories that look small but that impact unexpected areas of the system can come back to bite you. If your team forgets to consider all dependencies, and if the new code intersects with existing functionality, your story might take much longer than planned to finish. Make sure your story tests include the less obvious fallout from implementing the new functionality.

Take time to identify the central value each story provides and figure out an incremental approach to developing it. Plan small increments of writing tests, writing code, and testing the code some more. This way, your Quadrant 2 tests ensure you'll deliver the minimum value as planned.

THIN SLICES, SMALL CHUNKS

Writing stories is a tricky business. When the development team estimates new stories, it might find some stories too big, so it will ask the customer team to go back and break them into smaller stories. Stories can be too small as well, and might need to be combined with others or simply treated as tasks. Agile development, including testing, takes on one small chunk of functionality at a time.

When your team embarks on a new project or theme, ask the product owner to bring all of the related stories to a brainstorming session prior to the first iteration for that theme. Have the product owner and other interested stakeholders explain the stories. You might find that some stories need to be subdivided or that additional stories need to be written to fill in gaps.

After you understand what value each story should deliver and how it fits in the context of the system, you can break the stories down into small, manageable pieces. You can write customer tests to define those small increments, while keeping in mind the impact on the larger application.

A smart incremental approach to writing customer tests that guide development is to start with the "thin slice" that follows a happy path from one end to the other. Identifying a thin slice, also called a "steel thread" or "tracer bullet," can be done on a theme level, where it's used to verify the overall architecture. This steel thread connects all of the components together, and after it's solid, more functionality can be added.

See Chapter 10, "Business-Facing Tests that Critique the Product," for more about exploratory testing.

We find this strategy works at the story level, too. The sooner you can build the end-to-end path, the sooner you can do meaningful testing, get feedback, start automating tests, and start exploratory testing. Begin with a thin slice of the most stripped-down functionality that can be tested. This can be thought of as the critical path. For a user interface, this might start with simply navigating from one page to the next. We can show this to the customer and see whether the flow makes sense. We could write a simple automated GUI test. For the free-shipping threshold story at the beginning of this chapter, we might start by verifying the logic used to sum up the order total and determine whether it

See Part IV, "Automation," for more about regression test automation.

qualifies for free shipping, without worrying about how it will look on the UI. We could automate tests for it with a functional test tool such as FitNesse.

After the thin slice is working, we can write customer tests for the next chunk or layer of functionality, and write the code that makes those tests pass. Now we'll have feedback for this small increment, too. Maybe we add the UI to display the checkout page showing that the order qualified for free shipping, or add the layer to persist updates to the database. We can add on to the automated tests we wrote for the first pass. It's a process of "write tests—write code—run tests—learn." If you do this, you know that all of the code your team produces satisfies the customer and works properly at each stage.

Lisa's Story

My team has found that we have to focus on accomplishing a simple thin slice and add to it in tiny increments. Before we did this, we tended to get stuck on one part of the story. For example, if we had a UI flow that included four screens, we'd get so involved in the first one that we might not get to the last one, and there was no working end-to-end path. By starting with an end-to-end happy path and adding functionality a step at a time, we can be sure of delivering the minimum value needed.

Here's an example of our process. The story was to add a new conditional step to the process of establishing a company's retirement plan. This step allows users to select mutual fund portfolios, but not every user has access to this feature. The retirement plan establishment functionality is written in old, poorly designed legacy code. We planned to write the new page in the new architecture, but linking the new and old code together is tricky and error prone. We broke the story down into slices that might look tiny but that allowed us to manage risk and minimize the time needed to code and test the story. Figure 8-5 shows a diagram of incremental steps planned for this story.

The #1 thin slice is to insert a new, empty page based on a property. While it's not much for our customers to look at, it lets us test the bridge between old and new code, and then verify that the plan establishment navigation still works properly. Slice #2 introduces some business logic: If no mutual fund portfolios are available for the company, skip to the fund selection step, which we're not changing yet. If there are fund portfolios available, display them on the new step 3. In slice #3, we change the fund selection step, adding logic to display the funds that make up the portfolios. Slice #4 adds navigational elements between various steps in the establishment process.

We wrote customer tests to define each slice. As the programmers completed each one, we manually tested it and showed it to our customers. Any problems found were fixed immediately. We wrote an automated GUI test for slice #1, and added to it as the remaining steps were finished. The story was difficult because of the old legacy code interacting with the new architecture, but the stepwise approach made implementation smooth, and saved time.

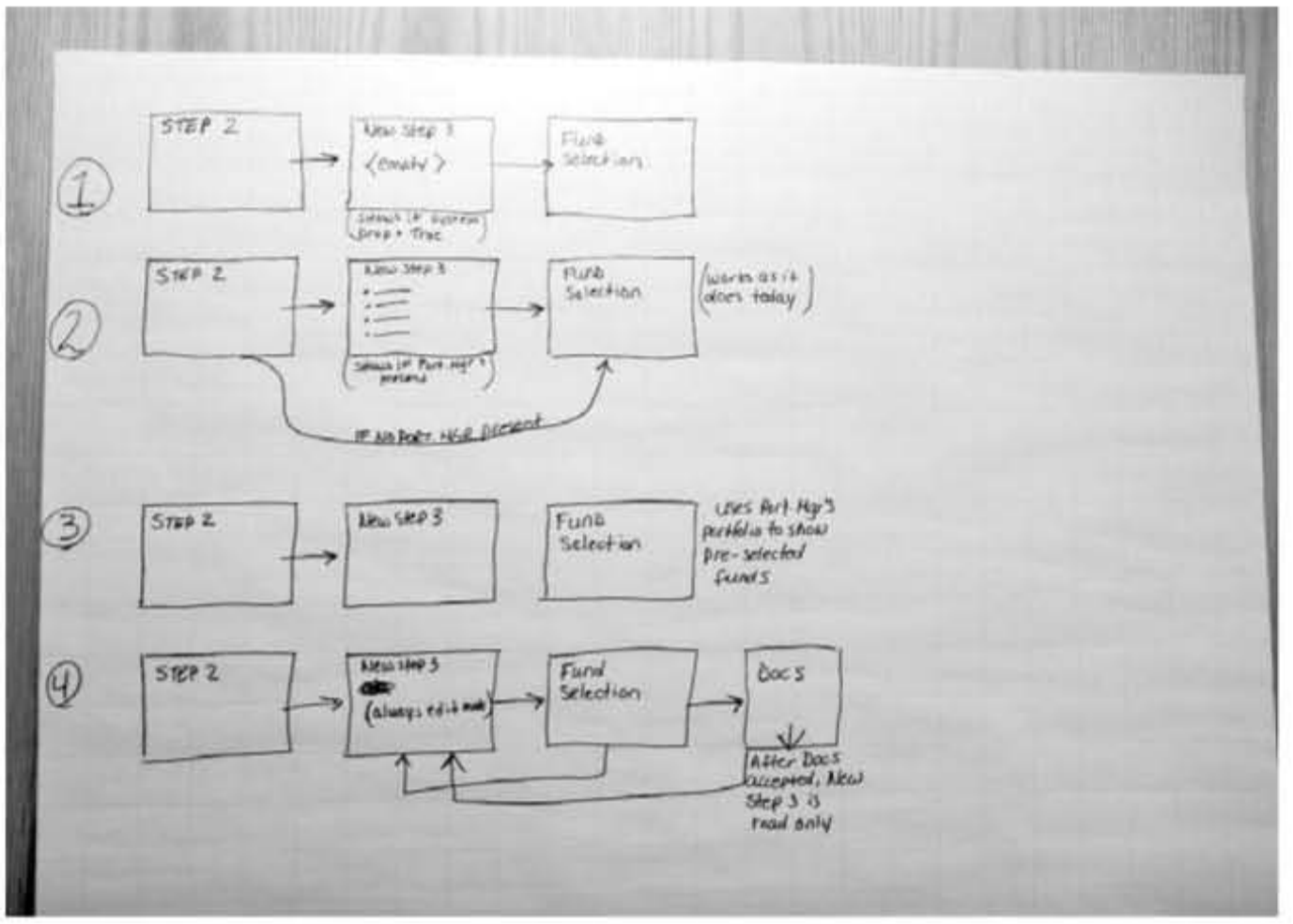


Figure 8-5 Incremental steps

When we draw diagrams such as this to break stories into slices, we upload photos of them to our team wiki so our remote team member can see them too. As each step is finished, we check it off in order to provide instant visual feedback.

—Lisa

Check the bibliography for Gerard Meszaros's article "Using Storyotypes to Split Bloated XP Stories."

If the task of writing customer tests for a story seems confusing or overwhelming, your team might need to break the story into smaller steps or chunks. Finishing stories a small step at a time helps spread out the testing effort so that it doesn't get pushed to the end of the iteration. It also gives you a better picture of your progress and helps you know when you're done—a subject we'll explore in the next section.

How Do We Know We're Done?

We have our business-facing tests that support the team—those tests that have been written to ensure the conditions of satisfaction have been met. They start with the happy path and show that the story meets the intended

need. They cover various user scenarios and ensure that other parts of the system aren't adversely affected. These tests have been run, and they pass (or at least they've identified issues to be fixed).

Are we done now? We could be, but we're not sure yet. The true test is whether the software's user can perform the action the story was supposed to provide. Activities from Quadrants 3 and 4, such as exploratory testing, usability testing, and performance testing will help us find out. For now, we just need to do some customer tests to ensure that we have captured all of the requirements. The business users or product owners are the right people to determine whether every requirement has been delivered, so they're the right people to do the exploring at this stage.

When the tests all pass and any missed requirements have been identified, we are done for the purpose of supporting the programmers in their quest for code that does the "right thing." It does not mean we are done testing. We'll talk much more about that in the chapters that follow.

Another goal of customer tests is to identify high-risk areas and make sure the code is written to solidify those. Risk management is an essential practice in any software development methodology, and testers play a role in identifying and mitigating risks.

TESTS MITIGATE RISK

Customer tests are written not only to define expected behavior of the code but to manage risk. Driving development with tests doesn't mean we'll identify every single requirement up front or be able to predict perfectly when we're done. It does give us a chance to identify risks and mitigate them with executable test cases. Risk analysis isn't a new technique. Agile development inherently mitigates some risks by prioritizing business value into small, tested deliverable pieces and by having customer involvement in incremental acceptance. However, we should still brainstorm potential events, the probability they might occur, and the impact on the organization if they do happen so that the right mitigation strategy can be employed.

Coding to predefined tests doesn't work well if the tests are for improbable edge cases. While we don't want to test only the happy path, it's a good place to start. After the happy path is known, we can define the highest risk scenarios—cases that not only have a bad outcome but also have a good possibility of happening.

In addition to asking the customer team questions such as "What's the worst thing that could happen?," ask the programmers questions like these: "What are the post conditions of this section of code? What should be persisted in the database? What behavior should we look for down the line?" Specify tests to cover potentially risky outcomes of an action.

Lisa's Story

My team considers worst-case scenarios in order to help us identify customer tests. For example, we planned a story to rewrite the first step of a multistep account creation wizard with a couple of new options. We asked ourselves questions such as the following: "When the user submits that first page, what data is inserted in the database? Are any other updates triggered? Do we need to regression test the entire account setup process? What about activities the user account might do after setup?" We might need to test the entire life cycle of the account. We don't have time to test more than necessary, so decisions about what to test are critical. The right tests help us mitigate the risk brought by the change.

—Lisa

Programmers can identify fragile parts of the code. Does the story involve stitching together legacy code with a new architecture? Does the code being changed interact with another system or depend on third-party software? By discussing potential impacts and risky areas with programmers and other team members, we can plan appropriate testing activities.

There's another risk. We might get so involved writing detailed test cases up front that the team loses the forest in the trees; that is, we can forget the big picture while we concentrate on details that might prove irrelevant.

Peril: Forgetting the Big Picture

It's easy to slip into the habit of testing only individual stories or basing your testing on what the programmer tells you about the code. If you find yourself finding integration problems between stories late in the release or that a lot of requirements are missing after the story is "done," take steps to mitigate this peril.

Always consider how each individual story impacts other parts of the system. Use realistic test data, use concrete examples as the basis of your tests, and have a lot of whiteboard discussions (or their virtual equivalent) in order to make sure everyone understands the story. Make sure the programmers don't start coding before any tests are written, and use exploratory testing to find gaps between stories.

Remember the end goal and the big picture.

As an agile team, we work in short iterations, so it's important to time-box the time spent writing tests before we start. After each iteration is completed, take the time to evaluate whether more detail up front would have helped. Were there enough tests to keep the team on track? Was there a lot of wasted time because the story was misunderstood? Lisa's team has found it best to write high-level story tests before coding, to write detailed test cases once coding starts, and then to do exploratory testing on the code as it's delivered in order to give the team more information and help make needed adjustments.

Janet worked on a project that had some very intensive calculations. The time spent creating detailed examples and tests before coding started, in order to ensure that the calculations were done correctly, was time well spent. Understanding the domain, and the impact of each story, is critical to assessing the risk and choosing the correct mitigation strategy.

While business-facing tests can help mitigate risks, other types of tests are also critical. For example, many of the most serious issues are usually uncovered during manual exploratory testing. Performance, security, stability, and usability are also sources of risk. Tests to mitigate these other risks are discussed in the chapters on Quadrants 3 and 4.

Experiment and find ways that your team can balance using up-front detail and keeping focused on the big picture. The beauty of short agile iterations is that you have frequent opportunities to evaluate how your process is working so that you can make continual improvements.

TESTABILITY AND AUTOMATION

When programmers on an agile team get ready to do test-driven development, they use the business-facing tests for the story in order to know what to code. Working from tests means that everyone thinks about the best way to design the code to make testing easier. The business-facing tests in Quadrant 2 are expressed as automated tests. They need to be clearly understood, easy to run, and provide quick feedback; otherwise, they won't get used.

It's possible to write manual test scripts for the programmers to execute before they check in code so that they can make sure they satisfied the customer's conditions, but it's not realistic to expect they'll go to that much trouble for long. When meaningful business value has to be delivered every two weeks or every 30 days, information has to be direct and automatic. Inexperienced agile teams might accept the need to drive coding with automated tests at the developer test level more easily than at the customer test

level. However, without the customer tests, the programmers have a much harder time knowing what unit tests to write.

Part IV, "Test Automation," will guide you as you develop an automation strategy.

Each agile team must find a process of writing and automating business-facing tests that drive development. Teams that automate only technology-facing tests find that they can have bug-free code that doesn't do what the customer wants. Teams that don't automate any tests will anchor themselves with technical debt.

Quadrant 2 contains a lot of different types of tests and activities. We need the right tools to facilitate gathering, discussing, and communicating examples and tests. Simple tools such as paper or a whiteboard work well for gathering examples if the team is co-located. More sophisticated tools help teams write business-facing tests that guide development in an executable, automatable format. In the next chapter, we'll look at the kinds of tools needed to elicit examples, and to write, communicate, and execute business-facing tests that support the team.

SUMMARY

In this chapter, we looked at ways to support the team during the coding process with business-facing tests.

- In agile development, examples and business-facing tests, rather than traditional requirements documents, tell the team what code to write.
- Working on thin slices of functionality, in short iterations, gives customers the opportunity to see and use the application and adjust their requirements as needed.
- An important area where testers contribute is helping customers express satisfaction conditions and create examples of desired, and undesired, behavior for each story.
- Ask open-ended questions to help the customer think of all of the desired functionality and to prevent hiding important assumptions.
- Help the customers achieve consensus on desired behavior for stories that accommodate the various viewpoints of different parts of the business.
- Help customers develop tools (e.g., a story checklist) to express information such as business satisfaction conditions.
- The development and customer teams should think through all of the parts of the application that a given story affects, keeping the overall system functionality in mind.

- Work with your team to break feature sets into small, manageable stories and paths within stories.
- Follow a pattern of "write test—write code—run tests—learn" in a step-by-step manner, building on each pass through the functionality.
- Use tests and examples to mitigate risks of missing functionality or losing sight of the big picture.
- Driving coding with business-facing tests makes the development team constantly aware of the need to implement a testable application.
- Business-facing tests that support the team must be automated for quick and easy feedback so that teams can deliver value in short iterations.