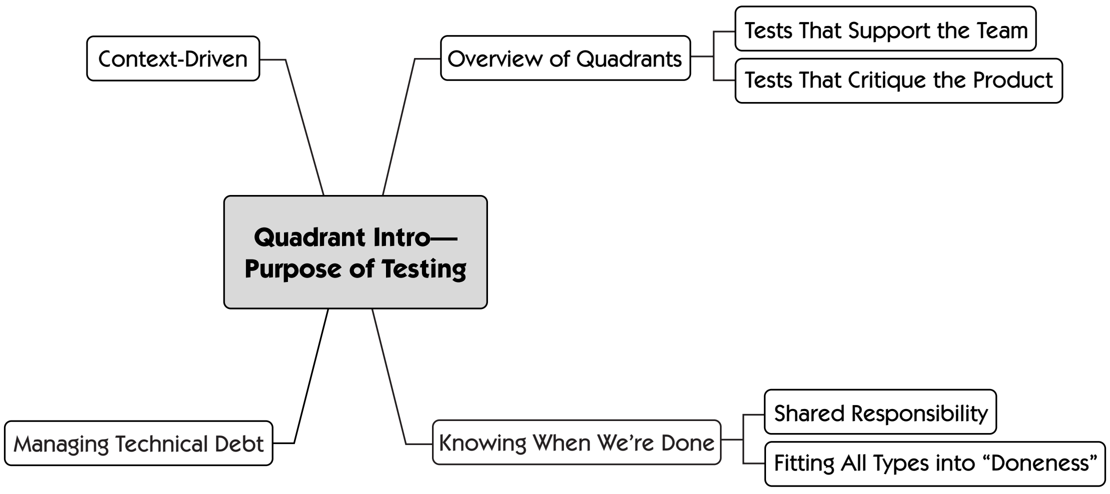
Chapter 6

THE PURPOSE OF TESTING


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

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

THE AGILE TESTING QUADRANTS

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

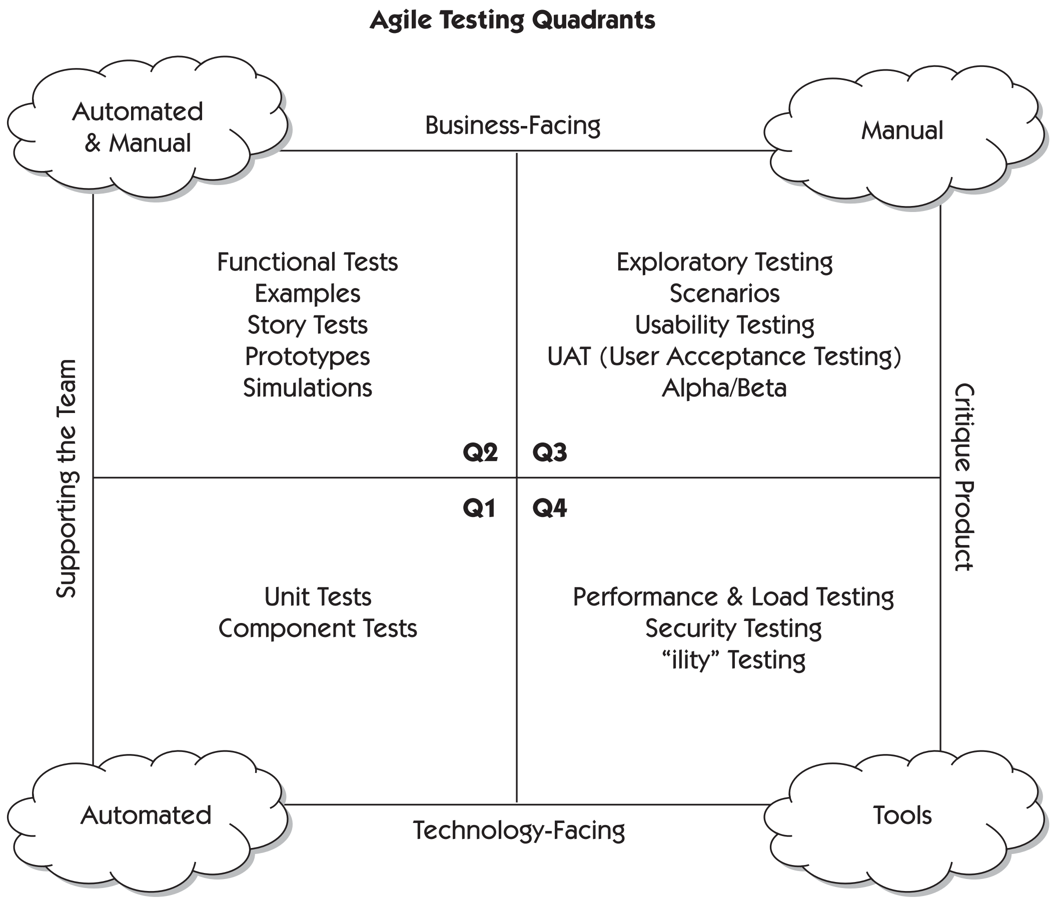


Figure 6-1 Agile Testing Quadrants

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

Tests that Support the Team

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

Quadrant 1

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

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

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

Quadrant 2

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

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

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

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

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

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

Using Tests to Support the Team

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

Lisa's Story

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

—Lisa

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

Tests that Critique the Product

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

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

Quadrant 3

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

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

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

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

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

Quadrant 4

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

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

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

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

Checking Nonfunctional Requirements

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

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

> Doing TDD gave us quick feedback on whether the code was working per the functional requirements, but the unit tests didn't test any 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.

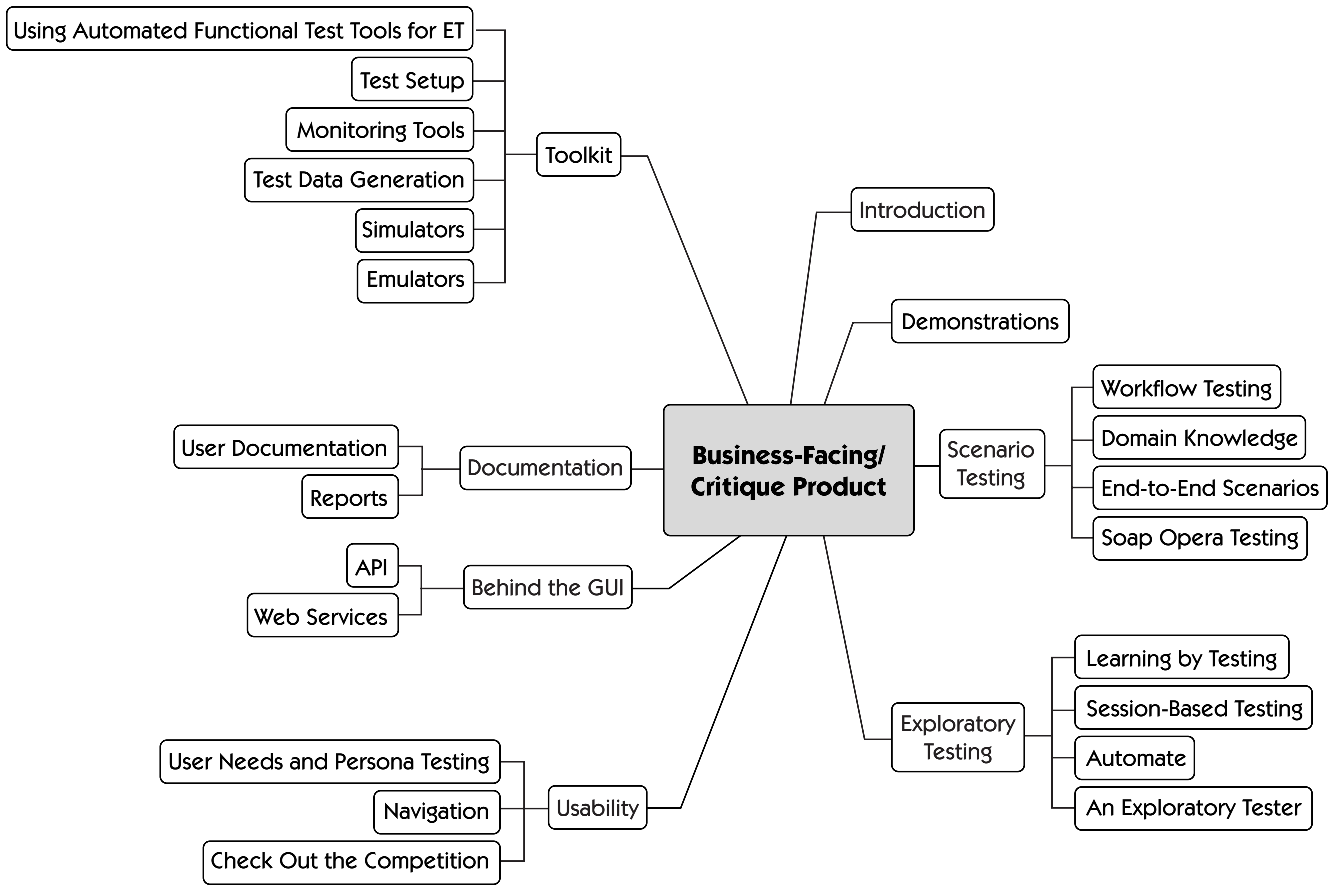
SUMMARY

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 10

Business-Facing Tests That Critique the Product



This chapter covers the third quadrant of the testing matrix. In Chapter 8, "Business-Facing Tests that Support the Team," we talked about the second quadrant and how to use business-facing tests to support programming. In this chapter, we show you how to critique the product with different types of business-facing tests. We'll also talk about tools that might help with these activities.

Introduction to Quadrant 3

Remember that business-facing tests are those you could describe in terms that would (or should) be of interest to a business expert. When we mention testing in traditional phased approaches, it pretty much always means critiquing the product after it is built. By now, you might think that in agile development this part of testing should be easy. After all, we just spent all that time making sure it works as expected. The requirements have all been tested as they were built, including security and other nonfunctional requirements, right? All that's left is to possibly find some obscure or interesting bugs.

As testers, we know that people make mistakes. No matter how hard we try to get it right the first time, we sometimes get it wrong. Maybe we used an example that didn't test what we thought it did. Or maybe we recorded a wrong expected result so the test passed, but it was a false positive. The business expert might have forgotten some things that real users needed. The best customer may not know what she wants (or doesn't want) until she sees it.

Critiquing or evaluating the product is what testers or business users do when they assess and make judgments about the product. These evaluators form perceptions based on whether they like the way it behaves, the look and feel, or the workflow of new screens. It is easier to see, feel, and touch a product and respond than to imagine what it will look like when it is described to you.

It's difficult to automate business-facing tests that critique the product, because such testing relies on human intellect, experience, and instinct. However, automated tools can assist with aspects of Quadrant 3 tests (see Figure 10-1), such as test data setup. The last section of this chapter contains examples of the types of tools that help teams focus on the important aspects of evaluating the product's value.

While much of the testing we discuss in this chapter is manual, don't make the mistake of thinking that this manual testing will be enough to produce high-quality software and that you can get away with not automating your regression tests. You won't have time to do any Quadrant 3 tests if you haven't automated the tests in Quadrants 1 and 2.

Evaluating or critiquing the product is about manipulating the system under test and trying to recreate actual experiences of the end users. Understanding different business scenarios and workflows helps to make the experience more realistic.

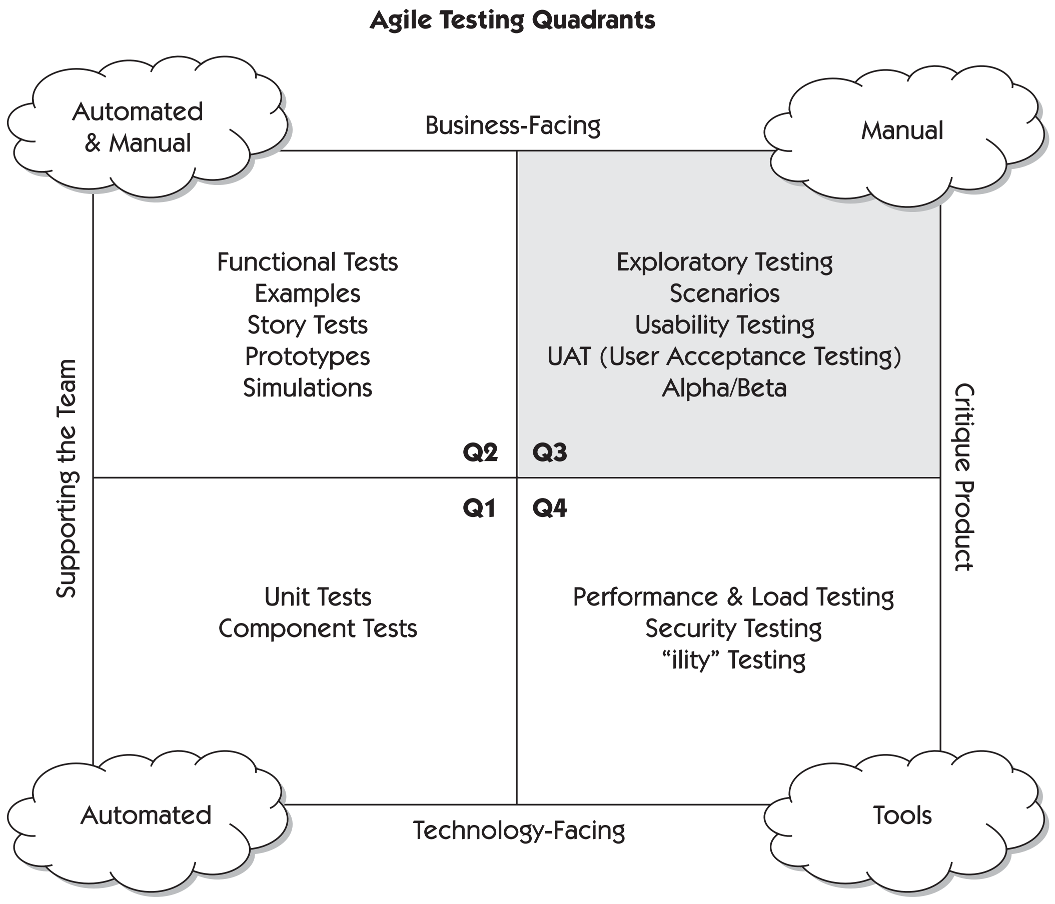


Figure 10-1 Quadrant 3 tests

DEMONSTRATIONS

We recommend showing customers what you're developing early and often. As soon as a rudimentary UI or report is available during story development, show it to the product owner or other domain expert on the team. However, not everyone on the business side will get a chance to see the iteration's deliverables until the iteration demo. End-of-iteration demonstrations are an opportunity for the business users and domain experts to see what has been delivered in the iteration and revise their priorities. It gives them a chance to say, "That's what I said, but it's not what I meant." This is a form of critiquing the product.

Janet's Story

I worked on a project that had five separate teams of eight to ten members, all developing the same system. Even though they were on the same floor, communication was an issue. There were many dependencies and overlaps, so the programmers depended on team-lead meetings to share information. However, the business users and testers needed to see what was being developed by other teams. They relied on end-of-iteration demonstrations given by each team to learn what the other teams were doing.

—Janet

Demonstrations to the executives or upper management can instill confidence in your project as well. One of the downfalls of a phased project is there is nothing to see until the very end, and management has to place all of its trust in the development team's reports. The incremental and iterative nature of agile development gives you a chance to demonstrate business value as you produce it, even before you release it. A live demonstration can be a very powerful tool if the participants are actively asking questions about the new features.

Rather than waiting until the end of the iteration, you can use any opportunity to demonstrate your changes. A recent project Janet worked on used regularly scheduled meetings with the business users to demonstrate new features in order to get immediate feedback. Any desired changes were fed into the next iteration.

Chapter 19, "Wrap Up the Iteration," talks about end-of-iteration demonstrations and reviews.

Choose a frequency for your demonstrations that works for your team so that the feedback loop is quick enough for you to incorporate changes into the release.

Informal demos can be even more productive. Sit down with a business expert and show her the story your team is currently coding. Do some exploratory testing together. We've heard of teams that get their stakeholders to do some exploratory testing after each iteration demo in order to help them think of refinements and future stories to change or build on the functionality just delivered.

Scenario Testing

Business users can help define plausible scenarios and workflows that can mimic end user behavior. Real-life domain knowledge is critical to creating accurate scenarios. We want to test the system from end to end but not necessarily as a black box.

One good technique for helping the team understand the business and user needs is "soap opera testing," a term coined by Hans Buwalda [2003]. The idea here is to take a scenario that is based on real life, exaggerate it in a manner similar to the way TV soap operas exaggerate behavior and emotions, and compress it into a quick sequence of events. Think about questions like, "What's the worst thing that can happen, and how did it happen?"

Soap Opera Test Example

Lisa worked on an Internet retail site, where she found soap opera tests to be effective. Here's an example of a soap opera scenario to test inventory, preorder, and backorder processes of an Internet retailer's warehouse.

> The most popular toy at our online toy store this holiday season is the Super Tester Action Figure. We have 20 preorders awaiting receipt of the items in our warehouse. Finally, Jane, a warehouse supervisor, receives 100 Super Tester Action figures. She updates the inventory system to show it is available inventory against the purchase order and no longer a preorder. Our website now shows Super Tester Action Figures available for delivery in time for the holidays. The system releases the preorders, which are sent to the warehouse. Meanwhile, Joe, the forklift driver, is distracted by his cell phone, and accidentally crashes into the shelf containing the Super Tester Action Figures. All appear to be smashed up beyond recognition. Jane, horrified, removes the 100 items from available inventory. Meanwhile, more orders for this popular toy have piled up in the system item. Sorting through the debris, Jane and Joe find that 14 of the action figures have actually survived intact. Jane adds them back into the available inventory.
>
> This scenario tests several processes in the system, including preorder, purchase order receipt, backorder, warehouse cancels, and preorder release. How many Super Tester toys will show as available on the shopping website at the end of all that? While executing the scenario, we'll probably find other areas we want to investigate; maybe the purchase order application is difficult to use or the warehouse inventory updates aren't reflected properly in the website. Thinking up and executing these types of tests will teach us more about what our users and other external customers need than running predefined functional tests on narrower areas of the application. As a bonus, it's fun!

Chapter 14, "An Agile Test Automation Strategy," examines different approaches to obtaining test data.

As a tester, we often "make up" test data, but it is usually simple so we can easily check our results. When testing different scenarios, both the data and the flow need to be realistic. Find out if the data comes from another system or if it's input manually. Get a sample if you can by asking the customers to provide data for testing. Real data will flow through the system and can be checked along the way. In large systems, it will behave differently depending on what decisions are made.

Tools to help define the scenarios and workflows can be simple. Data flow or process flow diagrams will help identify some of the common scenarios. These scenarios can help you think through a complex problem if you take the time. Consider the users and their motivation.

Lisa's Story

Our team planned to rewrite the core functionality of the application that processes the daily buys and sells of mutual funds. These trades are the result of retirement plan participants making contributions, exchanging balances from one fund to another, or withdrawing money from their accounts. Lisa's coworker, Mike Thomas, studied the existing trade processing flow and diagrammed it so that the team could understand it well before trying to rewrite the code. Figure 10-2 shows a portion of the flow diagram. WT stands for the custodian who does the actual trading. Three different file types are downloaded and translated into readable format: CFM, PRI, and POS. Each of these files feeds into a different part of the application to perform processing and produce various outputs: settled trades, a ticker exception report, and a fund position report.

—Lisa

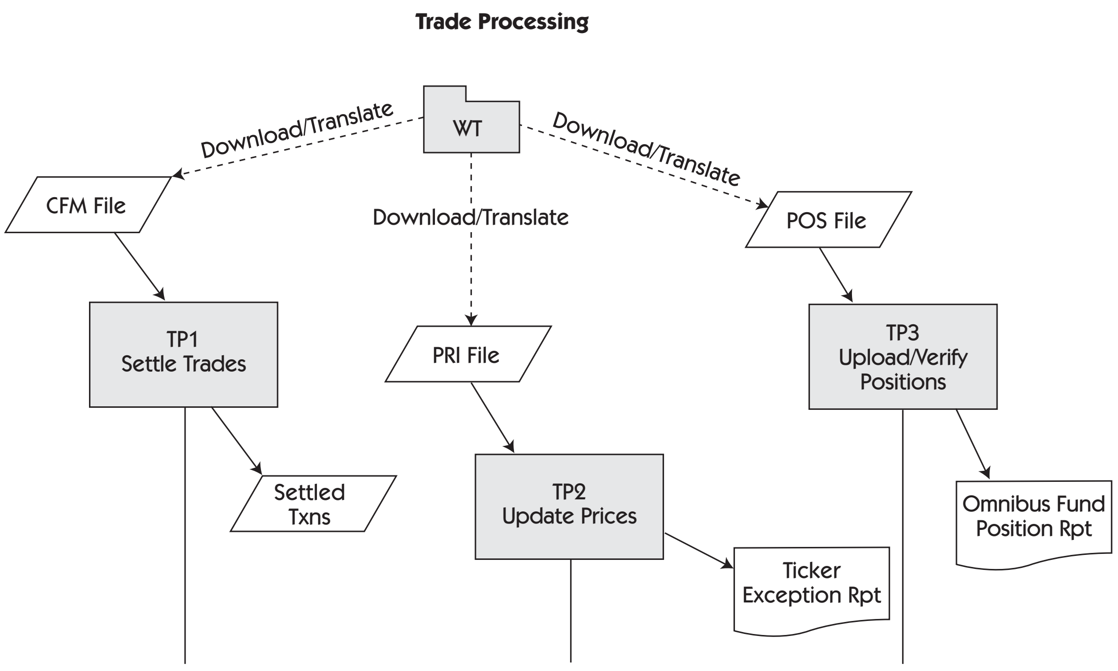


Figure 10-2 Sample portion of a process flow diagram

When testing end-to-end, make spot checks to make sure the data, status flags, calculations, and so on are behaving as expected. Use flow diagrams and other visual aids to help you understand the functionality. Many organizations depend on reports to make decisions, and those reports seem to be the last thing we verify. If your scenarios have been identified correctly, you might be able to use your application reports to provide a final check.

EXPLORATORY TESTING

Exploratory testing (ET) is an important approach to testing in the agile world. As an investigative tool, it's a critical supplement to the story tests and our automated regression suite. It is a sophisticated, thoughtful approach to testing without a script, and it enables you to go beyond the obvious variations that have already been tested. Exploratory testing combines learning, test design, and test execution into one test approach. We apply heuristics and techniques in a disciplined way so that the "doing" reveals more implications that just thinking about a problem. As you test, you learn more about the system under test and can use that information to help design new tests.

The bibliography lists more resources you should investigate to learn more about exploratory testing.

Exploratory testing is not a means of evaluating the software through exhaustive testing. It is meant to add another dimension to your testing. You do just enough to see if the "done" stories are really done to your satisfaction.

A valuable side effect of exploratory testing is the learning that comes out of it. It reveals areas of the product that could use more automated tests and brings up ideas for new or modified features that lead to new stories.

Exploratory Testing Explained

Michael Bolton is a trainer and consultant in rapid and exploratory testing approaches. He teaches a course called Rapid Software Testing, which he co-writes with senior author James Bach. Here's Michael's definition of exploratory testing.

See the bibliography for some links to more about Rapid Software Testing.

> Cem Kaner didn't invent exploratory testing, but he identified and named it in 1983, in the first edition of *Testing Computer Software*, as an approach all testers use when their brains are engaged in their work. He and James Bach, the other leading advocate of the approach, have long defined exploratory testing as "simultaneous test design, test execution, and learning." Kaner also defines exploratory testing more explicitly as "a style of testing that emphasizes the freedom and responsibility of the individual tester to continually optimize the value of her work by treating learning, test design, test execution, and test result interpretation as activities that continue in parallel throughout the project." That's quite a mouthful. What does it mean?

The most important thing to remember about exploratory testing is that it's not a test technique on its own. Instead, it's an approach or a mind-set that can be applied to any test technique. The second thing to remember is that exploratory testing is not merely about test execution; testers can also take an exploratory approach when they're designing new tests at the beginning of the iteration or analyzing the results of tests that have already been performed. A third important note is that exploratory testing isn't sloppy or slapdash or unprepared testing. An exploratory approach might require very extensive and elaborate preparation for certain tests—and an exploratory tester's knowledge and skill set, developed over years, is an often invisible yet important form of preparation. An exploratory test might be performed manually, or might employ extensive use of test automation—that is, any use of tools to support testing. So if exploratory testing isn't a technique, nor test execution, nor spontaneous, nor manual, what is it that makes a test activity exploratory? The answer lies in the cognitive engagement of the tester—how the tester responds to a situation that is continuously changing.

Suppose that a tester is given the mission to test a configuration dialog for a text editor. A tester using an exploratory approach would use specifications and conversations about the desired behavior to inform test ideas, but would tend to record these ideas in less detail than a tester using a scripted approach. A skilled tester doesn't generally need much explicit instruction unless the test ideas require some specific actions or data. If so, they might be written down or supplied to a program that could exercise them quickly. Upon seeing the dialog, the exploratory tester would interact with it, usually performing tests in accordance with the original test ideas—but she might also turn her attention to other ideas based on new problems or risks in the dialog as it appeared in front of her. Can two settings conflict in a way not covered by existing tests? The exploratory tester immediately investigates by performing a test on the spot. Does the dialog have a usability issue that could interfere with a user's work flow? The exploratory tester quickly considers a variety of users and scenarios and evaluates the significance of the problem. Is there a delay upon pressing the OK button? The exploratory tester performs a few more tests to seek a general pattern. Is there a possibility that some configuration options might not be possible on another platform? The exploratory tester notes the need for additional testing and moves on. Upon receiving new builds, the exploratory tester would tend to deemphasize repetition and emphasize variation in order to discover problems missed by older tests that are no longer revealing interesting information. This approach, which has always been fruitful, is even more powerful in environments where the need for repeated testing is handled by the developers' low-level, automated regression tests.

Exploratory testing is characterized by the degree to which the tester is under her own control, making informed choices about what he or she is going to do next, and where the last outcome of the last activity consciously informs the next choice. Exploratory and scripted approaches are at the opposite poles of a continuum. At the extreme end of the scripted mind-set, the decision as to what to do next comes exclusively from someone else, at some point in the past. In the exploratory mind-set, the decision to continue on the same line of inquiry or to choose a new path comes entirely from the individual tester, in the moment in which the activity occurs, The result of the last test strongly informs the tester's choices for the next test. Other influences include the stakeholders for whom test information might be important, the quality criteria that are important to stakeholders, the test coverage that stakeholders seek, specific risks associated with the item being tested, the needs of the end user of the product, the skills of the tester, the skills of the developers, the state of the item under test, the schedule for the project, the equipment and tools that are available to the tester, and the extent to which she can use them effectively—and that's only a partial list.

No test activity performed by a thinking human is entirely scripted. Humans have an extraordinary capacity to recognize things even when people are telling them not to, and as a result we can be distracted and diverted—but we can learn and adapt astonishingly quickly to new information and investigate its causes and effects. Machines only recognize what they've been programmed to recognize. When they're confronted with a surprising test result, at best they ignore it; at worst, they crash or destroy data.

Yet no test activity performed on behalf of a client is entirely exploratory, either. The exploratory tester is initially driven by the testing mission, which is typically set out by the client early in the project. Exploratory work can also be guided by checklists, strategy models, coverage outlines, risk lists—ideas that might come from other people at other times. The more that the tester is controlled by these ideas rather than guided by them, the more testing takes on a scripted approach.

Good exploration requires continuous investigation of the product by engaged human testers, in collaboration with the rest of the project community, rather than following a procedurally structured approach, performed exclusively by automation. *Exploration emphasizes individuals and interactions over processes and tools.* In an agile environment, where code is produced test-first and is covered with automated regression tests, testers can have not only the confidence but also the mandate to develop new tests and seek out new problems in the moment. *Exploration emphasizes responding to change versus following a plan.* Exploratory approaches use variation to drive an active search for problems instead of scripted manual or automated test cases that merely confirm what we already knew. *Exploration emphasizes working software over comprehensive documentation.* And to be effective, good

> exploration requires frequent feedback between testers, developers, customers, and the rest of the project community, not merely repetition of tests that were prepared at the beginning of the iteration, before we had learned important things about the project. *Exploration emphasizes customer collaboration over negotiated contracts.* Exploratory approaches are fundamentally agile.

Exploratory testing embraces the same values as agile development. It's an important part of the "agile testing mind-set" and critical to any team's success.

People unfamiliar with exploratory testing often confuse it with ad hoc testing. Exploratory testing isn't sitting down at a keyboard and typing away. Unskilled "black box" testers may not know how to do exploratory testing.

Exploratory testing starts with a charter of what aspects of the functionality will be explored. It requires critical thinking, interpreting the results, and comparing them to expectations or similar systems. Following "smells" when testing is an important component. Testers take notes during their exploratory testing sessions so that they can reproduce any issues they see and do more investigation as needed.

Technique: Exploratory Testing and Information Evaluation

Jon Hagar, an experienced exploratory tester, learner, and trainer, shares some activities, characteristics, and skills that are vital to effective exploratory testing.

> Exploratory testing uses the tester's understanding of the system, along with critical thinking, to define focused, experimental "tests" which can be run in short time frames and then fed back into the test planning process.
>
> An agile team has many opportunities to do exploratory testing, since each development cycle creates production-ready, working software. Starting early in each development cycle, consider exploratory tests based on:
>
> - Risk (analysis): The critical things you and the customer/user think can go wrong or be potential problems that will make people unhappy.
> - Models (mental or otherwise) of how software should behave: You and/or the customer have a great expectation about what the newly produced function should do or look like, so you test that.
> - Past experience: Think about how similar systems have failed (or succeeded) in predictable patterns that can be refined into a test, and explore it.

- What your development team is telling you: Talk to your developers and find out what "is important to us."
- Most importantly: What you learn (see and observe) as you test. As a tester on an agile team, a big part of your job is to constantly learn about your product, your team, and your customer. As you learn, you should quickly see tests based on such things as customer needs, common mistakes the team seems to be making, or good/bad characteristics of the product.

Some tests might be good candidates for automated regression suites. Some might just answer your exploratory charter and be "done." The agile team must critically think about what they are learning and "evolve" tests accordingly. The most important aspect here is to be "brain on" while testing, where you are looking for the "funny," unexpected, or new, which automated tests would miss. Use automation for what it is good at (repetitive tasks) and use agile humans for what we are good at (seeing, thinking, and dealing with the unexpected).

Several components are typically needed for useful exploratory testing:

- Test Design: An exploratory tester as a good test designer understands the many test methods. You should be able to call different methods into play on the fly during the exploration. This agility is a big advantage of exploratory testing over automated (scripted) procedures, where things must be thought out in advance.
- Careful Observation: Exploratory testers are good observers. They watch for the unusual and unexpected and are careful about assumptions of correctness. They might observe subtle software characteristics or patterns that drive them to change the test in real time.
- Critical Thinking: The ability to think openly and with agility is a key reason to have thinking humans doing nonautomated exploratory testing. Exploratory testers are able to review and redirect a test into unexpected directions on the fly. They should also be able to explain their logic of looking for defects and to provide clear status on testing. Critical thinking is a learned human skill.
- Diverse Ideas: Experienced testers and subject matter experts can produce more and better ideas. Exploratory testers can build on this diversity during testing. One of the key reasons for exploratory tests is to use critical thinking to drive the tests in unexpected directions and find errors.
- Rich Resources: Exploratory testers should develop a large set of tools, techniques, test data, friends, and information sources upon which they can draw. The agile test team members should grow their exploratory resources throughout a project and throughout their careers.

To help you understand a day in the life of an agile exploratory tester, here is a short tester's story:

> I arrived at 8:00 a.m. and reviewed what had happened the night before during automated testing. The previous night's automated tests found some minor but interesting errors. A password field on a login form had accepted a special character, which should have been rejected by the validation. I created an outline as a starting point for my "attack" (a top-level plan and/or risk list).
>
> As I thought about my "plan of attack," I sketched a small state model of the problem on a flip chart and showed this to a developer and my team's customer rep. I designed a test incorporating their suggestions, using some data stress inputs that I expected the validation to reject (a 1 MG file of special characters). I executed my test with my stress input, and, sure enough, the system rejected them as expected. I tried a different data set and the system failed with a buffer overflow in the database. I was learning, and we were on the trail of a potentially serious security bug. As the day went on, I explored different inputs to the password field and worked with the team to get the bug fixed.

You can learn from automated test results as well as from exploratory testing. Each type of testing feeds into the other. Develop a broad range of skills so you'll be able to identify important issues and write tests to prevent them from reoccurring.

The term *exploratory testing* was popularized by the "context-driven school" of testing. It's a highly disciplined activity, and it can be learned. Session-based test management is one method of testing that's designed to make exploratory testing auditable and measurable [Bach, 2003].

Session-Based Testing

Session-based testing combines accountability and exploratory testing. It gives a framework to a tester's exploratory testing experience so that they can report results in a consistent way.

Janet's Story

James Bach [2003] compares exploratory testing to putting together a jigsaw puzzle. When I first read his article with the jigsaw puzzle analogy, exploratory testing made perfect sense to me.

I start a jigsaw puzzle by dumping out all of the pieces of the puzzle and then sorting them into the different colors and edge pieces. Next, I put the edge pieces together, which gives me a framework in which to start. The edge of the jigsaw is analogous both to the mission statement, which helps me focus, and to the time-boxing of a session, which keeps me within certain limits.

Session-based testing is a form of exploratory testing, but it is time-boxed and a bit more structured. I learned about session-based testing from Jonathan Bach and found it gave me the structure I needed to do exploratory testing well. I use the same skills as I do for a jigsaw puzzle: I look for patterns in color or shapes or perhaps something that just doesn't look right, an anomaly. My thought process can take those patterns and make sense of them, using heuristics I have developed to help me solve a puzzle.

—Janet

Like solving the jigsaw puzzle by putting together the outside pieces first, we can use session-based testing to give us the framework in which we work. In session-based testing, we create a mission or a charter and then time-box our session so we can focus on what's important. Too often as testers, we can go off track and end up chasing a bug that might or might not be important to what we are currently testing.

For more information on session-based testing, check the bibliography for work by Jonathan Bach.

Sessions are divided into three kinds of tasks: test design and execution, bug investigation and reporting, and session setup. We measure the time we spend on setup versus actual test execution so that we know where we spend the most time. We can capture results in a consistent manner so that we can report back to the team.

Automation and Exploratory Testing

We can combine exploratory testing with test automation as well. Jonathan Kohl, in his article "Man and Machine" [2007], talks about interactive test automation to assist exploratory testing. Use automation to do test set up, data generation, repetitive tasks, or to progress along a workflow to the place you want to start. Then you start using your testing skills and experience to find the really "good" bugs, the insidious ones that otherwise escape attention. You can also use an automated test suite to explore. Just modify it a bit, watch the results as it runs, modify it again, and watch what happens.

An Exploratory Tester

With exploratory testing, each tester has a different approach to a problem, and has a unique style of working. However, there are certain attributes that make for a good exploratory tester. A good tester:

- Is systematic, but pursues "smells" (anomalies, pieces that aren't consistent)
- Learns to recognize problems through the use of Oracles (principle or mechanism by which we recognize a problem)

- Chooses a theme or role or mission statement to focus testing
- Time-boxes sessions and side trips
- Thinks about what the expert or novice user would do
- Explores together with domain experts
- Checks out similar or competitive applications

Exploratory testing helps us learn about the behavior of an application. Testers generally know a lot about the application they're testing. How do they judge whether the application is usable by users who are less technical or not familiar with it? Usability testing is vital for many software systems. We'll talk about that in the next section.

Usability Testing

There are two types of usability testing. The first type is the kind that is done up front by the user experience folks, using tools such as wire frames to help drive programming. Those types of tests belong in Quadrant 2. In this section, we're talking about the kind of usability testing that critiques the product. We use tools such as personas and our intuition to help us look at the product with the end user in mind.

User Needs and Persona Testing

Let's look at an online shopping example. We think about who will use the site. Will it be people who have shopped online before, or will it be brand new users who have no idea how to proceed? We're guessing it will be a mixture of both, as well as others. Take the time to ask your marketing group to get the demographics of the end users. The numbers might help you plan your testing.

One approach to using personas is for your team to invent several different users of your application representing different experience levels and needs. For our Internet retail application, we might have the following personas:

- Nancy Newbie, a senior citizen who is new to Internet shopping and nervous about identity theft
- Hudson Hacker, who looks for ways to cheat the checkout page
- Enrico Executive, who does all his shopping online and ships gifts to all his clients worldwide
- Betty Bargain, who's looking for great deals
- Debbie Ditherer, who has a hard time deciding what items she really wants to order

We might hang photos representing these different personas and their biographies in our work area so that we always keep them in mind. We can test the same scenario as each persona in turn and see what different experiences they might encounter.

Another way to approach persona testing, which we learned from Brian Marick and Elisabeth Hendrickson, is to pick a fictional character or famous celebrity and imagine how they would use our application. Would the Queen of England be able to navigate our checkout process? How might Homer Simpson search for the item he wants?

Real World Projects: Personas

The OneNote team at Microsoft uses personas as part of their testing process. Mike Tholfsen [2008], the Test Manager for OneNote, says they use seven personas that might use OneNote, specific customer types such as Attorneys, Students, Real Estate Agents, and Salespersons. The personas they create contain information such as:

- General job description
- "A Day in the Life"
- Primary uses for OneNote
- List of features the persona might use
- Potential notebook structures
- Other applications used
- Configuration and hardware environment

You can also just assume the roles of novice, intermediate, and expert users as you explore the application. Can users figure out what they are supposed to do without instructions? If you have a lot of first-time users, you might need to make the interface very simple.

Janet's Story

When I first started testing a new production accounting system, I found it very difficult to understand the flow, but the production accountants on the team loved it. After I worked with it for a while, I understood the complexity behind the application and knew why it didn't have to be intuitive for a first-time user. This was a good lesson for me, because I always assumed applications had to be user-friendly.

—Janet

If your application is custom-built for specific types of users, it might need to be "smart" rather than intuitive. Training sessions might be sufficient to get over the initial lack of usability so that the interface can be designed for maximum efficiency and utility.

Navigation

Navigation is another aspect of usability testing. It's incredibly important to test links and make sure the tabbing order makes sense. If a user has a choice of applications or websites, and has a bad first experience, they likely won't use your application again. Some of this testing is automatable, but it's important to test the actual user experience.

See Chapter 8, "Business-Facing Tests that Support the Team," for an example of Wizard of Oz testing, which is one approach to designing for usability.

If you have access to the end users, get them involved in testing the navigation. Pair with a real user, or watch one actually use the application and take notes. When you're designing a new user interface, consider using focus groups to evaluate different interfaces. You can start with mock-ups and flows drawn on paper, get opinions, and try HTML mock-ups next, to get early feedback.

Check Out the Competition

When evaluating your application for usability, think about other applications that are similar. How do they accomplish tasks? Do you consider them user-friendly or intuitive? If you can get access to competing software, take some time to research how those applications work and compare them with your product. For example, you're testing a user interface that takes a date range, and it has a pop-up calendar feature to select the date. Take a look at how a similar calendar function works on an airline reservation website.

See the bibliography for links to articles by Jeff Patton, Gerard Meszaros, and others on usability testing.

Usability testing is a fairly specialized field. If you're producing an internal application to be used by a few users who will be trained in its use, you probably don't need to invest much in usability testing. If you're writing the online directory assistance for a phone company, usability might be your main focus, so you need to learn as much as you can about it, or bring in a usability expert.

Chapter 9, "Toolkit for Business-Facing Tests that Support the Team," provides more detail about tools that facilitate these tests.

Behind the GUI

In a presentation titled "Man and Machine" [2007], Jonathan Kohl talked about alternatives for testing interfaces. Instead of always thinking about testing through the user interface, consider attacking the problem in other ways. Think about testing the whole system from every angle that you can approach. Consider using tools like simulators or emulators.

API Testing

In Chapter 8 and Chapter 9, we talked about testing behind the GUI to drive development. In this section, we show that you can extend your tests for the API in order to try different permutations and combinations.

An API (application programming interface) is a collection of functions that can be executed by other software applications or components. The end user is usually never aware that an API exists; she simply interacts with the interface on top.

Each API call has a specific function with a number of parameters that accept different inputs. Each variation will return a different result. The easy tests are simple inputs. The more complicated testing patterns occur when the parameters work together to give many possible variations. Sometimes parameters are optional, so it's important that you understand the possibilities. Boundary conditions should be considered as well, for both the inputs and expected results. For example, use both valid and invalid strings for parameters, vary the content, and vary the length of the strings' input.

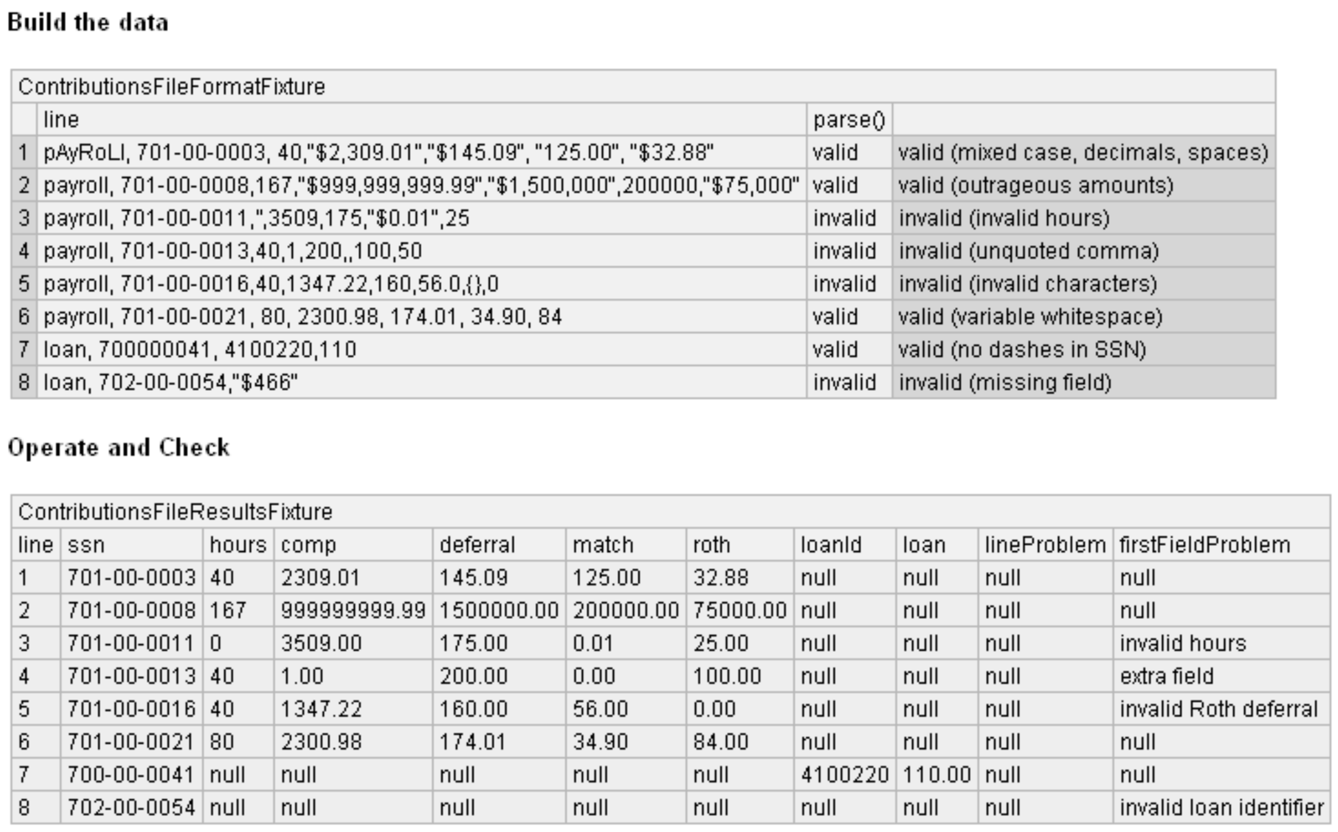
Another way to test is to vary the order of the API calls. Changing the sequence might produce unexpected results and reveal bugs that would never be found through UI testing. You can control the tests much more easily than when using the UI.

Lisa's Story

My team was working on a set of stories to enable retirement plan sponsors to upload payroll contribution files. We wrote FitNesse test cases to illustrate the file parsing rules, and the programmer wrote unit tests for those as well. When the coding for the parser was complete, we wanted to throw a lot more combinations of data at the parser, including some really bizarre ones, and see what happened. We could use the same fixture as we used for our tests to drive development, enter all of the crazy combinations we could think of, and see the results. We tested about 100 variations of both valid and invalid data. Figure 10-3 shows an example of just a few of the tests we tried. We found several errors in the code this way.

We didn't keep all of these tests in the regression suite because they were just a means of quickly trying every combination we could think of. We could have done these tests in a semi-automated, ad hoc manner too, not bothering to type the expected results into the result checking table, and just eyeballing the outputs to make sure they looked correct.

—Lisa



Build the data

ContributionsFileFormatFixture			
	line	parse()	
1	pAyRoLl, 701-00-0003, 40,"$2,309.01","$145.09", "125.00", "$32.88"	valid	valid (mixed case, decimals, spaces)
2	payroll, 701-00-0008,167,"$999,999,999.99","$1,500,000",200000,"$75,000"	valid	valid (outrageous amounts)
3	payroll, 701-00-0011,",3509,175,"$0.01",25	invalid	invalid (invalid hours)
4	payroll, 701-00-0013,40,1,200,,100,50	invalid	invalid (unquoted comma)
5	payroll, 701-00-0016,40,1347.22,160,56.0,{},0	invalid	invalid (invalid characters)
6	payroll, 701-00-0021, 80, 2300.98, 174.01, 34.90, 84	valid	valid (variable whitespace)
7	loan, 700000041, 4100220,110	valid	valid (no dashes in SSN)
8	loan, 702-00-0054,"$466"	invalid	invalid (missing field)

Operate and Check

ContributionsFileResultsFixture										
line	ssn	hours	comp	deferral	match	roth	loanId	loan	lineProblem	firstFieldProblem
1	701-00-0003	40	2309.01	145.09	125.00	32.88	null	null	null	null
2	701-00-0008	167	999999999.99	1500000.00	200000.00	75000.00	null	null	null	null
3	701-00-0011	0	3509.00	175.00	0.01	25.00	null	null	null	invalid hours
4	701-00-0013	40	1.00	200.00	0.00	100.00	null	null	null	extra field
5	701-00-0016	40	1347.22	160.00	56.00	0.00	null	null	null	invalid Roth deferral
6	701-00-0021	80	2300.98	174.01	34.90	84.00	null	null	null	null
7	700-00-0041	null	null	null	null	null	4100220	110.00	null	null
8	702-00-0054	null	null	null	null	null	null	null	null	invalid loan identifier

Figure 10-3 Sample of parsing rules test

Janet's Story

I recently worked with a web application that interfaces to a legacy system through a well-defined API. Due to the design of the legacy system and the fact that the data is hard to replicate, the team hasn't yet found a way to automate this testing. However, we could look in the log files to verify the correct inputs were passed and the expected result was returned. Valuable exploratory testing of APIs is possible with or without benefit of automation.

—Janet

API calls can be developed early in an application life cycle, which means testing can occur early as well. Testing through an API can give confidence in the system before a UI is ever developed. Because this type of testing can be automated, you will need to work with your programmers to understand all of the parameters and the purpose of each function. If your programmers or automation team develop a test harness that is easy to use, you should be able to methodically create a suite of test cases that exercises the functionality.

Web Services

Web services generally require plenty of security, stress, and reliability testing. See Chapter 12, "Critiquing the Product using Technology-Facing Tests," for more on these types of tests.

Web services are a services-based architecture that provides an external interface so that others can access the system. There might be multiple stakeholders, and you may not even know who will be using your product. Your testing will need to confirm the quality of service that the external customers expect.

Consider levels of service that have been promised to clients when you are creating your test plans. Make time for exploratory testing to simulate the different ways users might access the web services.

The use of web services standards also offers other implications for current testing tools. As with API calls, web services-based integration highlights the importance of validating interface points. However, we also need to consider message formats and processing, queuing times, and message response times.

Chapter 12, "Summary of Testing Quadrants," has an example of testing web services.

Using testing tools that utilize GUI-driven automation is simply inadequate for a web services project. A domain-specific language that encapsulates implementation details "behind the scenes" works well for testing web services.

Testing Documents and Documentation

One of the components of the system that is often overlooked during testing is documentation. As agile developers, we may value working software over documentation, but we still value documentation! User manuals and online help need validation just as much as software. Your team may employ specialists such as technical writers who create and verify documentation. As with all other components of the product, your whole team is responsible for the quality of the documentation, and that includes both hard copy and electronic.

User Documentation

Your team might do Quadrant 2 tests to support the team as they produce documentation; in fact we encourage it. Lisa's team writes code that produces documents whose contents are specified by government regulations, and programmers can write much of the code test-first. However, it's difficult for automated tests to judge whether a document is formatted correctly or uses a readable font. They also can't evaluate whether the contents of documents such as user manuals are accurate or useful. Because documentation has many subjective components, validating it is more of a critiquing activity.

Janet's Story

Technical writers and testers can work very closely together. Stephanie, a technical writer I worked with on one project, talked with the programmers to understand how the application worked. She would also work through the application to make sure she wrote it down correctly. This seemed to be a duplication of the testing effort, so Stephanie and I sat down and figured out a better approach.

We decided to work together on the stories as they were developed. For some stories Stephanie was lead "tester," and sometimes I took that role. If I was lead, I'd create my test conditions and examples and Stephanie would use those as her basis for the documentation. When Stephanie was lead, she would write her documentation, and then I would use that to determine the test cases.

Doing it this way enabled the documentation to be tested and the tests to be challenged before they were ever executed. Working hand in hand like this proved to be a very successful experiment. The resulting documentation matched the software's behavior and was much more useful to the end users.

—Janet

Don't forget to check the help text too. Are the links to help text easily identifiable? Are they consistent throughout the user interface? Is the help text presented clearly? If it opens in a pop-up, and users block pop-ups in their browsers, what's the impact? Does the help cover all of the topics needed? On Lisa's projects, help text tends to be a low priority, so it often doesn't get done at all. That's a business decision, but if you feel an area of the application needs extra help text or documentation, raise the issue to your team and your customers.

Reports

Another system component that's often overlooked from a testing perspective is reports. Reports are critical to many users for decision-making purposes but are often left until the very end, and either don't get done or are poorly executed. Reports might be tailored to meet specific customer needs, but there are many third-party tools available for generating reports. Reports may be part of the application itself or be generated through a separate reporting system for end users.

We discuss testing reports along with the other Quadrant 3 test activities in order to critique the product, but we recommend that you also write Quadrant 2 report tests that will guide the coding and help the team understand the customer's needs as it produces reports. They can certainly be written test-first. Like documents, though, you need to look at a report to know if it's easy enough to read and presents information in an understandable way.

One of the biggest challenges when testing reports is not the formatting but getting the right data. When you try to create test data for reports, it can be difficult to get a good cross section of realistic data. It also is usually the edge cases that make the reports fail, so incorporating that extra data is not feasible. In most cases, it's best to use production data (or data copied from the production system into a test environment) to test the different reporting variations.

Lisa's Story

Our application includes a number of reports, many of which help companies meet governmental compliance requirements. While we have automated smoke tests for each report, any change to a report, or even an upgrade in the tool we use to generate reports, requires extensive manual and visual testing. We have to watch like hawks: Has a number been truncated by one character? Did a piece of text run over to the next page? Is the right data included? Wrong or missing data can mean trouble with the regulatory agency.

Another challenge is verifying the data contained in the report. If I were to use the same query that the report uses, it doesn't prove anything. I sometimes struggle to come up with my own SQL queries to compare the actual data with what shows up on a report. We budget extra time to test reports, even the simple-looking ones.

Because reports are so subjective, we find that different stakeholders have different preferences for how the data is presented. The plan administrator who has to explain a report to a user on the phone has a different idea of what's easy to understand than the company lawyer who decides what data needs to be on the report. Our product owner helps us get consensus from all areas of the business.

The contents and formatting of a report are important, of course, but for online reports, the speed at which they come up is critical too. Our plan administrators wanted complete freedom to specify any date range for some transaction history reports. Our DBA, who coded the reports, warned that for a large company's retirement plan, data for more than a few months worth of transactions could take several minutes to render. Over time, companies grew, they had more and more transactions, and eventually the user interface started timing out before it could deliver the report. When testing, try out worst-case scenarios, which could eventually become the most common scenario.

—Lisa

See Chapter 8, "Business-Facing Tests that Support the Team," for information about using thin slices.

If you're tackling a project that involves lots of reports, don't give in to the temptation to leave them to the end. Include some reports in each iteration if you can. One report could be a single story or maybe even broken up into a couple of stories. Use mock-ups to help the customers decide on report contents and formatting. Find the "thin slice" or "critical path" in the report, code that first, and show it to your customer before you add the next slice. Incremental development works as well with reports as it does with other software.

Sometimes your customers themselves aren't sure how a report should look or how to approach it incrementally. And sometimes nobody on the team anticipates how hard the testing effort will prove to be.

Lisa's Story

Like other financial accounts, retirement plans need to provide periodic statements to account holders that detail all of the money going into and out of the account. These statements show the change in value between the beginning and ending balances and other pertinent information, such as the names of account beneficiaries. Our company wanted to improve the account statements, both as a marketing tool and to reduce the number of calls from account holders who didn't understand their statements.

We didn't have access to our direct competitors' account statements, so the product owner asked for volunteers to bring in account statements from banks and other financial institutions in order to get ideas. Months of discussions and experimentation with mock-ups produced a new statement format, which included data that wasn't on the report previously, such as performance results for each mutual fund.

Stories for developing the new account statement were distributed throughout two quarters worth of iterations. During the first quarter, stories to collect new data were done. Testing proved much harder than we thought. We used FitNesse tests to verify capturing the different data elements, which lulled us into a false sense of security. It was hard to cover all of the variations, and we missed some with the automated tests. We also didn't anticipate that the changes to collect new data could have an adverse effect on the data that already displayed on the existing statements.

As a result, we didn't do adequate manual testing of the account statements. Subtle errors slipped past us. When the job to produce quarterly statements ran, calls started coming in from customers. We had a mad scramble to diagnose and fix the errors in both code and data. The whole project was delayed by a quarter while we figured out better ways to test and added internal checks and better logging to the code.

—Lisa

Short iterations mean that it can be hard to make time for adequate exploratory testing and other Quadrant 3 activities. Let's look at tools that might help speed up this testing and make time for vital manual and visual tests.

Tools to Assist with Exploratory Testing

Exploratory testing is manual testing. Some of the best testing happens because a person is paying attention to details that often get missed if we are following a script. Intuition is something that we cannot make a machine learn. However, there are many tools that can assist us in our quest for excellence.

Tools shouldn't replace human interaction; they should enhance the experience. Tools can provide testers with more power to find the hard-to-reproduce bugs that often get filed away because no one can get a handle on them. Exploratory testing is unconventional, so why shouldn't the tools be as well? Think about low-effort, high-value ways that tools can be incorporated into your testing.

See the bibliography for references to Jonathan Kohl's writings on using human and automation power together for optimal testing.

Computers are good at doing repetitive tasks and performing calculations. These are two areas where they are much better than humans, so let's use them for those tasks. Because testing needs to keep pace with coding, any time advantage we can gain is a bonus.

In the next few sections, we'll look at some areas where automation can leverage exploratory testing. The ones we cover are test setup, test data generation, monitoring, simulators, and emulators.

Test Setup

Let's think about what we do when we test. We've just found a bug, but not one that is easily reproducible. We're pretty sure it happens as a result of interactions between components. We go back to the beginning and try one scenario after another. Soon we've spent the whole day just trying to reproduce this one bug.

Ask yourself how you can make this easier. We've found that one of the most time-consuming tasks is the test setup and getting to the right starting point for your actual test. If you use session-based testing, then you already know how much time you spend setting up the test, because you have been tracking that particular time waster. This is an excellent opportunity for some automation.

The tools used for business-facing tests that support the team described in Chapter 9 are also valuable for manual exploratory testing. Automated functional test scripts can be run to set up data and scenarios to launch exploratory testing sessions. Tests configured to accept runtime parameters are particularly powerful for setting up a starting point for evaluating the product.

Lisa's Story

Our Watir test scripts all accept a number of runtime parameters. When I need a retirement plan with a specific set of options, and specific types of participants, I can kick off a Watir script or two with some variables set on the command line. When the scripts stop, I have a browser session with all of the data I need for testing already set up. This is so fast that I can test permutations I'd never get to using all-manual keystrokes.

—Lisa

The test scripts you use for functional regression testing and for guiding development aren't the only tools that help take the tedium out of manual exploratory testing. There are other tools to help set up test data as well as to help you evaluate the outputs of your testing sessions.

Whatever tool you are using, think about how it can be adapted to run the scenario over and over with different inputs plugged in. Janet has also successfully used Ruby with Watir to set up tests to run multiple times to help identify bugs. Tools that drive the browser or UI in much the same way that an end user would makes your testing more reliable because you can play it back on your monitor and watch for anything that might not look as it should during the setup. When you get to the place where the test actually starts, you can then use your excellent testing abilities to track down the source of the bug.

Test Data Generation

PerlClip is an example of a tool that you can use to test a text field with different kinds of inputs. James Bach provides it free of charge on his website, www.satisfice.com, and it can be very helpful in validating fields. For example, if you have a field that will accept a maximum input of 200 characters, testing this field and its boundaries manually would be very tedious. Use PerlClip to create a string, put it in your automation library, and have your automation tool call the string to test the value.

Monitoring Tools

Tools like the Unix/Linux command `tail -f`, or James Bach's LogWatch, can help monitor log files for error conditions. IDEs also provide log analysis tools. Many error messages are never displayed on the screen, so if you're testing via the GUI, you never see them. Get familiar with tools like these, because they can make your testing more effective and efficient. If you are not sure where your system logs warnings and errors, ask your developers. They probably have lots of ideas about how you can monitor the system.

Simulators

See the "System Test" example in Chapter 12, "Summary of Testing Quadrants," to see how a simulator was critical to the testing the whole system.

Simulators are tools used to create data that represent key characteristics and behavior of real data for the system under test. If you do not have access to real data for your system, simulated data will sometimes work almost as well. The other advantage of using a simulator is for pumping data into a system over time. It can be used to help generate error conditions that are difficult to create under normal circumstances and can reduce time in boundary testing.

Setting up data and test scenarios is half of the picture. You also need to have a way to watch the outcomes of your testing. Let's consider some tools for that purpose.

Emulators

An emulator duplicates the functionality of a system so that it behaves like the system under test. There are many reasons to use an emulator. When you need to test code that interfaces with other systems or devices, emulators are invaluable.

Two Examples of Emulators

WestJet, a Canadian airline company, provides the capability for guests to use their mobile devices to check in at airports that support the feature. When testing this application, it is better for both the programmers and the testers to test various devices as early as possible. To make this feasible, they use downloadable emulators to test the Web Check-in application quickly and often during an iteration. Real devices, which are expensive to use, can then be used sparingly to verify already tested functionality.

The team also created another type of emulator to help test against the legacy system being interfaced with. The programmers on the legacy system have different priorities and delivery schedules, and a backlog of requests. To prevent this from holding up new development, the programmers on the web application have created a type of emulator for the API into the legacy system that returns predetermined values for specific API calls. They develop against this emulator, and when the real changes are available, they test and make any modifications then. This change in process has enabled them to move ahead much more quickly than was previously possible. It has proved to be a simple but very powerful tool.

Emulators are one tool that helps to keep testing and coding moving together hand-in-hand. Using them is one way for testing to keep up with development in short iterations. As you plan your releases and iterations, think about the types of tools that might help with creating production-like test scenarios. See if you can use the tools you're already using for automating tests to drive development as aids to exploratory testing.

Driving development with tests is critical to any project's success. However, we humans won't always get all of the requirements for desired system behavior entirely correct. Our business experts themselves can miss important aspects of functionality or interaction with other parts of the system when they provide examples of how a feature should work. We have to use techniques to help both the customer and developer teams learn more about the system so they can keep improving the product.

Summary

A large part of the testing effort is spent critiquing the product from a business perspective. This chapter gave you some ideas about the types of tests you can do to make your testing efforts more effective.

- Demonstrate software to stakeholders in order to get early feedback that will help direct building the right stuff.
- Use scenarios and workflows to test the whole system from end to end.
- Use exploratory testing to supplement automation and to take advantage of human intellect and perceptions.
- Without usability in mind when testing and coding, applications can become shelfware. Always be aware of how the system is being used.
- Testing behind the GUI is the most effective way of getting at the application functionality. Do some research to see how you can approach your application.
- Incorporate all kinds of tests to make a good regression suite.
- Don't forget about testing documentation and reports.
- Automation tools can perform tedious and repetitive tasks, such as data and test scenario setup, and free up more time for important manual exploratory testing.
- Tools you're already using to automate functional tests might also be useful to leverage exploratory tests.

- Monitoring, resource usage, and log analysis tools built into operating systems and IDEs help testers appraise the application's behavior.
- Simulators and emulators enable exploratory testing even when you can't duplicate the exact production environment.
- Even when tests are used to drive development, requirements for desired behavior or interaction with other systems can be missed or misunderstood. Quadrant 3 activities help teams keep adding value to the product.