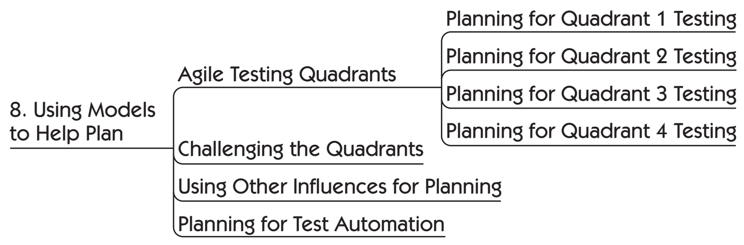
Chapter 8

Using Models to Help Plan



8. Using Models to Help Plan
Agile Testing Quadrants
Planning for Quadrant 1 Testing
Planning for Quadrant 2 Testing
Planning for Quadrant 3 Testing
Planning for Quadrant 4 Testing
Challenging the Quadrants
Using Other Influences for Planning
Planning for Test Automation

As agile development becomes increasingly mainstream, there are established techniques that experienced practitioners use to help plan testing activities in agile projects, although less experienced teams sometimes misunderstand or misuse these useful approaches. Also, the advances in test tools and frameworks have somewhat altered the original models that applied back in the early 2000s. Models help us view testing from different perspectives. Let's look at some foundations of agile test planning and how they are evolving.

Agile Testing Quadrants

The agile testing quadrants (the Quadrants) are based on a matrix Brian Marick developed in 2003 to describe types of tests used in Extreme Programming (XP) projects (Marick, 2003). We've found the Quadrants to be quite handy over the years as we plan at different levels of precision. Some people have misunderstood the purpose of the Quadrants. For example, they may see them as sequential activities instead of a taxonomy of testing types. Other people disagree about which testing activities belong in which quadrant and avoid using the Quadrants altogether. We'd like to clear up these misconceptions.

Figure 8-1 is the picture we currently use to explain this model. You'll notice we've changed some of the wording since we presented it in *Agile*

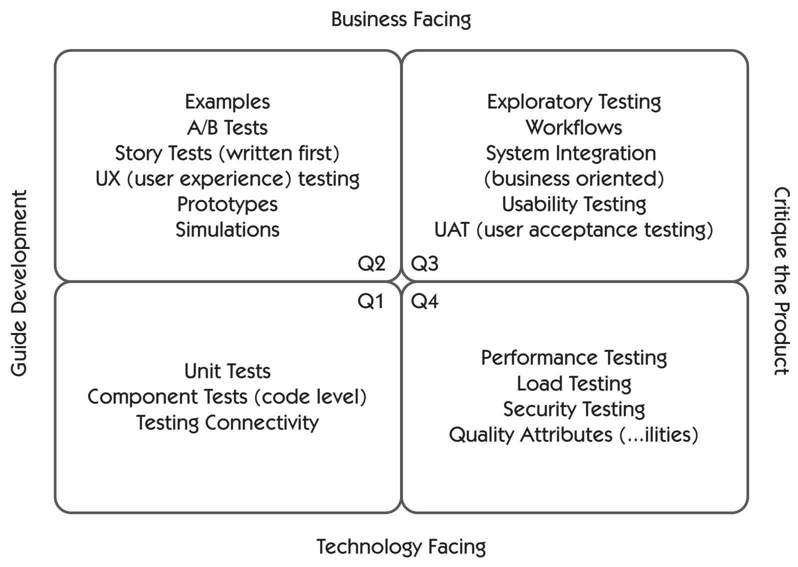


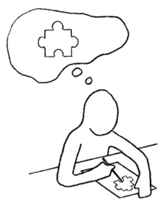
Figure 8-1 Agile testing quadrants

Testing. For example, we now say "guide development" instead of "support development." We hope this makes it clearer.

It's important to understand the purpose behind the Quadrants and the terminology used to convey their concepts. The quadrant numbering system does *not* imply any order. You don't work through the quadrants from 1 to 4, in a sequential manner. It's an arbitrary numbering system so that when we talk about the Quadrants, we can say "Q1" instead of "technology-facing tests that guide development." The quadrants are

- **Q1:** technology-facing tests that guide development
- **Q2:** business-facing tests that guide development
- **Q3:** business-facing tests that critique (evaluate) the product
- **Q4:** technology-facing tests that critique (evaluate) the product

The left side of the quadrant matrix is about preventing defects before and during coding. The right side is about finding defects and discovering missing features, but with the understanding that we want to find them as fast as possible. The top half is about exposing tests to the business, and the bottom half is about tests that are more internal to the team but equally important to the success of the software product. "Facing" simply refers to the language of the tests—for example, performance tests satisfy a business need, but the business would not be able to read the tests; they are concerned with the results.



Most agile teams would start with specifying Q2 tests, because those are where you get the examples that turn into specifications and tests that guide coding. In his 2003 blog posts about the matrix, Brian called Q2 and Q1 tests "checked examples." He had originally called them "guiding" or "coaching" examples and credits Ward Cunningham for the adjective "checked." Team members would construct an example of what the code needs to do, check that it doesn't do it yet, make the code do it, and check that the example is now true (Marick, 2003). We include prototypes and simulations in Q2 because they are small experiments to help us understand an idea or concept.

In some cases it makes more sense to start testing for a new feature using tests from a different quadrant. Lisa has worked on projects where her team used performance tests for a spike for determination of the architecture, because that was the most important quality attribute for the feature. Those tests fall into Q4. If your customers are uncertain about their requirements, you might even do an investigation story and start with exploratory testing (Q3). Consider where the highest risk might be and where testing can add the most value.

Most teams concurrently use testing techniques from all of the quadrants, working in small increments. Write a test (or check) for a small chunk of a story, write the code, and once the test is passing, perhaps automate more tests for it. Once the tests (automated checks) are passing, use exploratory testing to see what was missed. Perform security or load testing, and then add the next small chunk and go through the whole process again.

Michael Hüttermann adds "outside-in, barrier-free, collaborative" to the middle of the quadrants (see Figure 8-2). He uses behavior-driven development (BDD) as an example of barrier-free testing. These tests are written in a natural, ubiquitous "given_when_then" language that's accessible to customers as well as developers and invites conversation

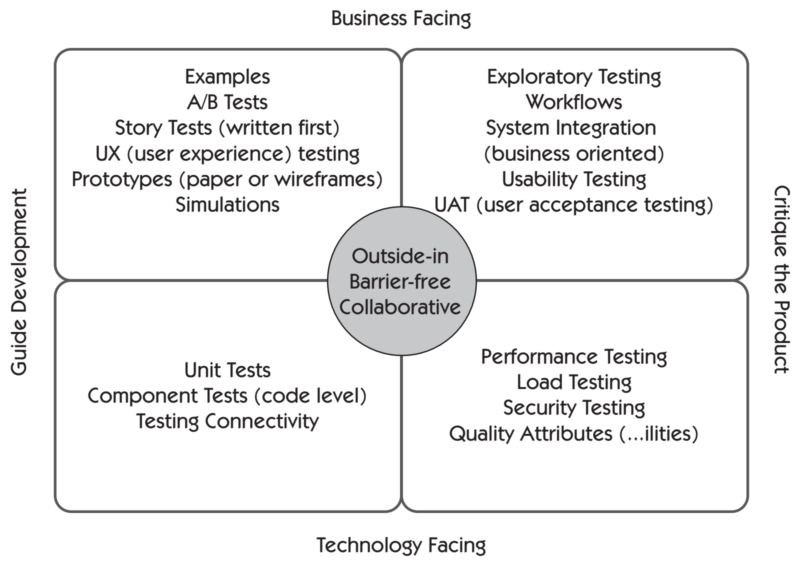


Figure 8-2 Agile testing quadrants (with Michael Hüttermann's adaptation)

between the business and the delivery team. This format can be used for both Q1 and Q2 checking. See Michael's *Agile Record* article (Hüttermann, 2011b) or his book *Agile ALM* (Hüttermann, 2011a) for more ideas on how to augment the Quadrants.

The Quadrants are merely a taxonomy or model to help teams plan their testing and make sure they have all the resources they need to accomplish it. There are no hard-and-fast rules about what goes in which quadrant. Adapt the Quadrants model to show what tests your team needs to consider. Make the testing visible so that your team thinks about testing first as you do your release, feature, and story planning. This visibility exposes the types of tests that are currently being done and the number of people involved. Use it to provoke discussions about testing and which areas you may want to spend more time on.

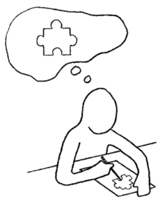
When discussing the Quadrants, you may realize there are necessary tests your team hasn't considered or that you lack certain skills or resources to be able to do all the necessary testing. For example, a team that Lisa worked on realized that they were so focused on turning business-facing examples into Q2 tests that guide development that they were completely ignoring the need to do performance and security testing. They added in user stories to research what training and tools they would need and then budgeted time to do those Q4 tests.

Planning for Quadrant 1 Testing

Back in the early 1990s, Lisa worked on a waterfall team whose programmers were required to write unit test plans. Unit test plans were definitely overkill, but thinking about the unit tests early and automating all of them were a big part of the reason that critical bugs were never called in to the support center. Agile teams don't plan Q1 tests separately. In test-driven development (TDD), also called test-driven design, testing is an inseparable part of coding. A programmer pair might sit and discuss some of the tests they want to write, but the details evolve as the code evolves. These unit tests guide development but also support the team in the sense that a programmer runs them prior to checking in his or her code, and they are run in the CI on every single check-in of code.

There are other types of technical testing that may be considered as guiding development. They might not be obvious, but they can be critical to keeping the process working. For example, let's say you can't do your testing because there is a problem with connectivity. Create a test script that can be run before your smoke test to make sure that there are no technical issues. Another test programmers might write is one to check the default configuration. Many times these issues aren't known until you start deploying and testing.

Planning for Quadrant 2 Testing



Q2 tests help with planning at the feature or story level. Part IV, "Testing Business Value," will explore guiding development with more detailed business-facing tests. These tests or checked examples are derived from collaboration and conversations about what is important to the feature or story. Having the right people in a room to answer questions and give specific examples helps us plan the tests we need. Think about the levels of precision discussed in the preceding chapter; the questions and the examples get more precise as we get into details about the stories. The process of eliciting examples and creating tests from them fosters collaboration across roles and may identify defects in the form of hidden assumptions or misunderstandings before any code is written.

Show everyone, even the business owners, what you plan to test; see if you're standing on anything sacred, or if they're worried you're missing something that has value to them.

Creating Q2 tests doesn't stop when coding begins. Lisa's teams have found it works well to start with happy path tests. As coding gets under way and the happy path tests start passing, testers and programmers flesh out the tests to encompass boundary conditions, negative tests, edge cases, and more complicated scenarios.

Planning for Quadrant 3 Testing

Testing has always been central to agile development, and guiding development with customer-facing Q2 tests caught on early with agile teams. As agile teams have matured, they've also embraced Q3 testing, exploratory testing in particular. More teams are hiring expert exploratory testing practitioners, and testers on agile teams are spending time expanding their exploratory skills.

Planning for Q3 tests can be a challenge. We can start defining test charters before there is completed code to explore. As Elisabeth Hendrickson explains in her book *Explore It!* (Hendrickson, 2013), charters let us define where to explore, what resources to bring with us, and what information we hope to find. To be effective, some exploratory testing might require completion of multiple small user stories, or waiting until the feature is complete. You may also need to budget time to create the user personas that you might need for testing, although these may already have been created in story-mapping or other feature-planning exercises. Defining exploratory testing charters is not always easy, but it is a great way to share testing ideas with the team and to be able to track what testing was completed. We will give examples of such charters in Chapter 12, "Exploratory Testing," where we discuss different exploratory testing techniques.

One strategy to build in time for exploratory testing is writing stories to explore different areas of a feature or different personas. Another strategy, which Janet prefers, is having a task for exploratory testing for each story, as well as one or more for testing the feature. If your team uses a definition of "done," conducting adequate exploratory testing might be part of that. You can size individual stories with the assumption that you'll spend a significant amount of time doing exploratory testing. Be aware that unless time is specifically allocated during task creation, exploratory testing often gets ignored.

Q3 also includes user acceptance testing (UAT). Planning for UAT needs to happen during release planning or as soon as possible. Include your customers in the planning to decide the best way to proceed. Can they come into the office to test each new feature? Perhaps they are in a different country and you need to arrange computer sharing. Work to get the most frequent and fastest feedback possible from all of your stakeholders.

Planning for Quadrant 4 Testing

Quadrant 4 tests may be the easiest to overlook in planning, and many teams tend to focus on tests to guide development. Quadrant 3 activities such as UAT and exploratory testing may be easier to visualize and are often more familiar to most testers than Quadrant 4 tests. For example, more teams need to support their application globally, so testing in the internationalization and localization space has become important. Agile teams have struggled with how to do this; we include some ideas in Chapter 13, "Other Types of Testing."

Some teams talk about quality attributes with acceptance criteria on each story of a feature. We prefer to use the word *constraints*. In *Discover to Deliver* (Gottesdiener and Gorman, 2012), Ellen Gottesdiener and Mary Gorman recommend using Tom and Kai Gilb's Planguage (their planning language; see the bibliography for Part III, "Planning—So You Don't Forget the Big Picture," for links) to talk about these constraints in a very definite way (Gilb, 2013).

If your product has a constraint such as "Every screen must respond in less than three seconds," that criterion doesn't need to be repeated for every single story. Find a mechanism to remind your team when you are discussing the story that this constraint needs to be built in and must be tested. Liz Keogh describes a technique to write tests about how capabilities such as system performance can be monitored (Keogh, 2014a). Organizations usually know which operating systems or browsers they are supporting at the beginning of a release, so add them as constraints and include them in your testing estimations. These types of quality attributes are often good candidates for testing at a feature level, but if it makes sense to test them at the story level, do so there; think, "Test early." Chapter 13, "Other Types of Testing," will cover a few different testing types that you may have been struggling with.

Challenging the Quadrants

Over the years, many people have challenged the validity of the Quadrants or adjusted them slightly to be more meaningful to them. We decided to share a couple of these stories because we think it is valuable to continuously challenge what we "know" to be true. That is how we learn and evolve to improve and meet changing demands.

Gojko's Challenge to the Quadrants

Gojko Adzic, *an author and strategic software delivery consultant, challenges the validity of the Quadrants in the current software delivery era.*

The agile testing quadrants model is probably the one thing that everyone remembers about the original *Agile Testing* book. It was an incredibly useful thinking tool for the software delivery world then—2008. It helped me facilitate many useful discussions on the big picture missing from typical programmers' view of quality, and it helped many testers figure out what to focus on. The world now, as of 2014, looks significantly different. There has been a surge in the popularity of continuous delivery, DevOps, Big Data analytics, lean startup delivery, and exploratory testing. The Quadrants model is due for a serious update.

One of the problems with the original Quadrants model is that it was easily misunderstood as a sequence of test types—especially that there is some kind of division between things before and things after development.

This problem is even worse now than in 2008. With the surge in popularity of continuous delivery, the dividing line is getting more blurred and is disappearing. With shorter iterations and continuous delivery, it's generally difficult to draw the line between activities that support the team and those that critique the product. Why would performance tests not be aimed at supporting the team? Why are functional tests not critiquing the product? Why is UAT separate from functional testing? I always found the horizontal dimension of the Quadrants difficult to justify, because critiquing the product can support the team quite effectively if it is done in a timely way. For example, specification by example helps teams to completely merge functional tests and UAT into something that is continuously checked during development. Many teams I worked with recently run performance tests during development, primarily not to mess things up with frequent changes. These are just two examples where things on the right side of the Quadrants are now used more to support the team than anything else. With lean startup methods, products get a lot of critiquing even before a single line of production code is written.

Dividing tests into those that support development and those that evaluate the product does not really help to facilitate useful discussions anymore, so we need a different model—in particular, one that helps to address the eternal issue of so-called nonfunctional requirements, which for many people actually means, "It's going to be a difficult discussion, so let's not have it." The old Quadrants model puts "ilities" into a largely forgotten quadrant of technical tests after development. But things like security, performance, scalability, and so on are not really technical; they imply quite a lot of business expectations, such as compliance, meeting service-levels agreements, handling expected peak loads, and so on. They are also not really nonfunctional, as they imply quite a lot of functionality such as encryption, caching, and work distribution. This of course is complicated by the fact that some expectations in those areas are not that easy to define or test for—especially the unknown unknowns. If we treat these as purely technical concerns, the business expectations are often not explicitly stated or verified. Instead of nonfunctional, these concerns are often dysfunctional. And although many "ilities" are difficult to prove before the software is actually in contact with its real users, the emergence of A/B split testing techniques over the last five years has made it relatively easy, cheap, and low risk to verify those things in production.

Another aspect of testing not really captured well by the first book's Quadrants is the surge in popularity and importance of exploratory testing. In the old model, exploratory testing is something that happens from the business perspective in order to evaluate the product (often misunderstood as after development). In many contexts, well documented in Elisabeth Hendrickson's book on exploratory testing (Hendrickson, 2013) and James Whittaker's book *How Google Tests Software* (Whittaker et al., 2012), exploratory testing can be incredibly useful for the technical perspective as well and, more importantly, is something that should be done during development.

The third aspect that is not captured well by the early Quadrants is the possibility to quantify and measure software changes through usage analytics in production. The surge in popularity of Big Data analytics, especially combined with lean startup and continuous delivery models, enables teams to test relatively cheaply things that were very expensive to test ten years ago—for example, true performance impacts. When the original *Agile Testing* book came out, serious performance testing often meant having a complete hardware copy of the production system. These days, many teams de-risk those issues with smaller, less risky continuous changes, whose impact is measured directly on a subset of the production environment. Many teams also look at their production log trends to spot unexpected and previously unknown problems quickly.

We need to change the model (Figure 8-3) to facilitate all those discussions, and I think that the current horizontal division isn't helping anymore. The context-driven testing community argues very forcefully that looking for expected results isn't really testing; instead, they call that checking. Without getting into an argument about what is or isn't testing, I found the division to be quite useful for many recent discussions with clients. Perhaps that is a more useful second axis for the model: the difference between looking for expected outcomes and analyzing unknowns, aspects without a definite yes/no answer, where results require skillful analytic interpretation. Most of the innovation these days seems to happen in the second part anyway. Checking for expected results, from both a technical and business perspective, is now pretty much a solved problem.

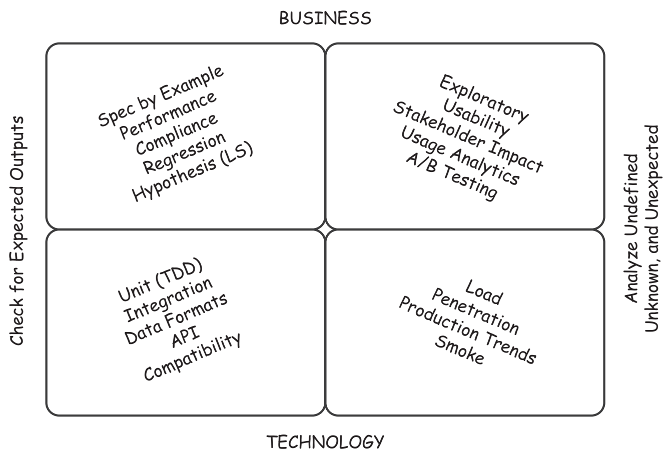


Figure 8-3 Gojko Adzic's version of the agile testing quadrants

Thinking about checking expected outcomes versus analyzing outcomes that weren't predefined helps to explain several important issues facing software delivery teams today:

Security concerns could be split easily into functional tests for compliance such as encryption, data protection, authentication, and so on (essentially all checking for predefined expected results), and penetration/investigations (not predefined). This will help to engage the delivery team and business sponsors in a more useful discussion about describing the functional part of security up front.

Performance concerns could be divided into running business scenarios to prove agreed-upon service levels and capacity, continuous delivery style (predefined), and load tests (where will it break?). This will help to engage the delivery team and business in defining performance expectations and prevent people from treating performance as a purely technical concern. By avoiding the support the team/evaluate the product divisions, we allow a discussion of executing performance tests in different environments and at different times.

Exploration would become much more visible and could be clearly divided between technical and business-oriented exploratory tests. This can support a discussion of technical exploratory tests that developers should perform or that testers can execute by reusing existing automation frameworks. It can also support an overall discussion of what should go into business-oriented exploratory tests.

Build-measure-learn product tests would fit into the model nicely, and the model would facilitate a meaningful discussion of how those tests require a defined hypothesis and how that is different from just pushing things out to see what happens through usage analytics.

We can facilitate a conversation on how to spot unknown problems by monitoring production logs as a way of continuously testing technical concerns that are difficult to check and expensive to automate before deployment, but still useful to support the team. By moving the discussion away from supporting development or evaluating the product toward checking expectations or inspecting the unknown, we would also have a nice way of differentiating those tests from business-oriented production usage analytics.

Most importantly, by using a different horizontal axis, we can raise awareness about a whole category of things that don't fit into typical test plans or test reports but are still incredibly valuable. The early Quadrants were useful because they raised awareness about a whole category of things in the upper-left corner that most teams weren't really thinking of but are now taken as common sense. The 2010s Quadrants need to help us raise awareness about some more important issues for today.

	✓ CONFIRM	INVESTIGATE
BUSINESS	BUSINESS-FACING EXPECTATIONS	RISKS TO EXTERNAL QUALITY ATTRIBUTES
TECHNOLOGY	TECHNOLOGY-FACING EXPECTATIONS	RISKS TO INTERNAL QUALITY ATTRIBUTES

Figure 8-4 Elisabeth Hendrickson's version of the agile testing quadrants

Elisabeth Hendrickson also presented an alternative to the existing Quadrants in her talk about "The Thinking Tester" (Hendrickson, 2012). It is similar to Gojko's version but has a different look. You can see in Figure 8-4 that she relabeled the vertical columns to "confirm" and "investigate," while the horizontal rows still represent business and technology.

The top left quadrant represents the expectations of the business, which could be in the form of executable (automated) specifications. Others might be represented by paper prototypes or wireframes. At the top right are tests that help investigate risks concerning the external quality of the product. It is very much like the original quadrant's idea of exploratory testing, scenarios, or usability testing. Like Gojko's model, the bottom right quadrant highlights the risks of the internal working of the system.

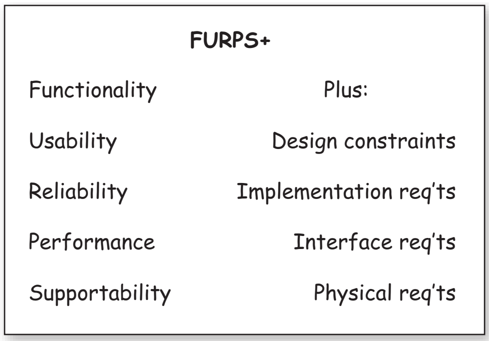
Both of these alternative models provide value. We think there is room for multiple variations to accommodate a spectrum of needs. For example, organizations that are able to adopt continuous delivery are able to think in this space, but many organizations are years from accomplishing that. Check the bibliography for Part III for links to additional testing quadrant models. Use them to help make sure your team covers all the different types of tests you need in order to deliver the right value for your customers.

Using Other Influences for Planning

There are many useful models and ideas for helping us in our test planning, and we shouldn't throw them away. As Tim Ottinger and Jeff Langr have said (Ottinger and Langr, 2009b), a mnemonic for thinking about what are called nonfunctional requirements is still useful. The FURPS model (see Figure 8-5) was developed at Hewlett-Packard and was first publicly elaborated by Grady and Caswell (Wikipedia, 2014f); it is now widely used in the software industry. The + was later added to the model after various campaigns at HP to extend the acronym to emphasize various attributes.

James Whittaker developed a methodology he calls the Attribute Component Capability (ACC) matrix (Whittaker, 2011) to help define what to test based on risk. ACC consists of three different parts that define the system under test: Attributes, Components, and Capabilities. He defines these as:

- **Attributes** (adjectives of the system) are qualities and characteristics that promote the product and distinguish it from the competition; examples are "Fast," "Secure," "Stable," and "Elegant."
- **Component**s (nouns of the system) are building blocks that together constitute the system in question. Some examples of



FURPS+

Functionality	Plus:
Usability	Design constraints
Reliability	Implementation req'ts
Performance	Interface req'ts
Supportability	Physical req'ts

Figure 8-5 FURPS+ flash card (Ottinger and Langr, 2011)

Components are "Firmware," "Printing," and "File System" for an operating system project, or "Database," "Cart," and "Product Browser" for an online shopping site.

- **Capabilitie**s (verbs of the system) describe the abilities of a particular Component to satisfy the Attributes of the system. An example Capability for a shopping site could be "Processes monetary transactions using HTTPS." You can see that this could be a Capability of the "Cart" component when trying to meet the "Secure" Attribute. The most important aspect of Capabilities is that they are testable.

Creating a high-level matrix using this model can be a simple way to visualize your system. Figure 8-6 shows an example of what such a matrix might look like. Gojko Adzic agrees that exposing system characteristics and providing more visibility is definitely a good idea (Adzic, 2010a), though he cautions that while we can learn from other fields, we should be careful about using them as a metaphor for software development.

Use heuristics such as Elisabeth Hendrickson's "Test Heuristics Cheat Sheet" (Hendrickson, 2011) or tried-and-true techniques such as state diagrams or truth tables to think of new ideas for attributes. Combine these ideas with models like the Quadrants so that the conversations about the system constraints or usability can extract clear examples. Using all the tools in your toolbox can only help increase the quality of the product.

Components			Capabilities	Attributes		
Mobile App	Firmware	Printing		Fast	Secure	Stable
			Manage profile			
			Send messages			
			Update network			
INFLUENCE AREA				RISK / IMPORTANCE		

Figure 8-6 ACC example

Planning for Test Automation

Since Mike Cohn came up with his test automation pyramid in 2003, many teams have found it a useful model to plan their test automation. To take advantage of fast feedback, we need to consider at what level our automation tests should be. When we look at the standard pyramid, Figure 8-7, we see three levels.

The lowest level is the base—the unit tests. When we consider testing, we should try to push the tests as low as they can go for the highest return on investment (ROI) and the quickest feedback.

However, when we have business logic where tests need to be visible to the business, we should use collaborative tools that create tests at the service layer (the API) to specify them in a way that documents system behavior. See Chapter 16, "Test Automation Design Patterns and Approaches," for

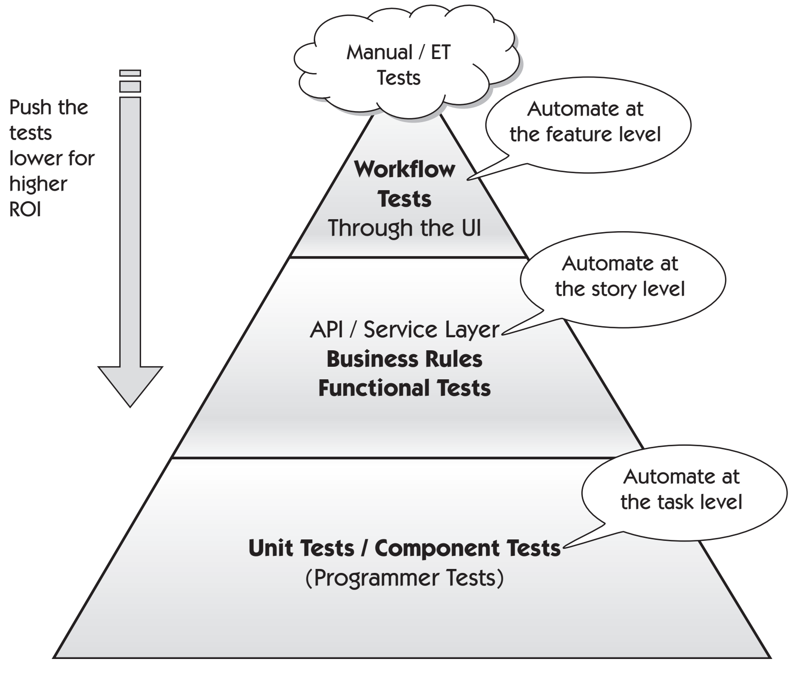

Manual / ET Tests
Push the tests lower for higher ROI
Automate at the feature level
Workflow Tests Through the UI
Automate at the story level
API / Service Layer Business Rules Functional Tests
Automate at the task level
Unit Tests / Component Tests (Programmer Tests)

Figure 8-7 Automation pyramid

more details. It is at this layer that we can automate at the story level so that testing and automation can keep up with the coding.

The top layer of the pyramid consists of the workflow tests through the user interface (UI). If we have a high degree of confidence in the unit tests and the service-level or API-level tests, we can keep these slower, more brittle automated tests to a minimum. See Chapter 15, "Pyramids of Automation," for more detail on alternative pyramid models.

Practices such as guiding development with examples can help define what the best level for the test is. A team's cadence can be set by how well they plan and execute their automation and how well they understand the level of detail they need. Consider also how to make your automation test runs visible, whether displayed in the continuous integration environment or on a monitor that is in the open.

Summary

Models are a useful tool for planning. In this chapter, we covered the following points:

- The agile testing quadrants provide a model for thinking about testing in an agile world.
  - The Quadrants help to emphasize the whole-team responsibility for testing.
  - They provide a visible mechanism for talking about the testing needed.
  - The left side is about guiding development, learning what to build, and preventing defects—testing early.
  - The right side is about critiquing the product, finding defects, and learning what capabilities are still missing.
- Gojko Adzic provides an alternative way to think about the Quadrants if you are in a lean startup or continuous delivery environment.
- We also introduced an alternative quadrant diagram from Elisabeth Hendrickson that highlights confirmatory checks versus investigative testing.

- There are already many tools in our agile testing toolbox, and we can combine them with other models such as the Quadrants to make our testing as effective as possible.
- FURPS and ACC are additional examples of models you can use to help plan based on risk and a variety of quality characteristics.
- The automation pyramid is a reminder to think about automation and to plan for it at the different levels.