Chapter 9. The Role of Quality Assurance in Lean-Agile Software Development

When you are up to your ass in alligators, it's hard to remember your original intention was to drain the swamp. —Author unknown.

In This Chapter

This chapter covers several critical issues surrounding the role of quality assurance in Lean-Agile software development:

- The role of testers must be one of preventing defects, not finding them
- How moving the specification of acceptance test to the start of the development cycle can greatly reduce waste—both the waste of building the wrong thing and the waste of building the thing wrong.
- What to do when it is not easy to test early.

Note

The terms "Quality Assurance" (QA) and "Quality Control" (QC) are used in a variety of ways in our industry. IT organizations and product organizations, in particular, seem to have different meanings for the same term.

For purposes of this chapter,

- Quality Control is the practice of ensuring products or serves are designed and produced to meet or exceed customer requirements and
- Quality Assurance refers to planned and systematic production processes that provide confidence in a product's suitability for its intended purpose.

We recognize that, in some organizations, QA refers to ensuring that people are following the process they are supposed to. That is not what we mean in this chapter. Here, QA focuses on ensuring that the product is both suitable for the customer and that it is built correctly.

Takeaways

Key insights to take away from this chapter include:

- Testers should look to avoid problems not fix them
- Moving testing up front can improve quality at little, if any, extra cost.
- Developers should always ask the question – "how will I know I've done that?" before writing code

- We strongly endorse up-front testing, but if you are not going to do it, we feel you should at least consider tests before writing code.

Introduction

Two Lean principles factor prominently in quality assurance: *build quality in* and *eliminate waste.* Defects indicate that quality is not being built in to the product. And a lack of quality causes a lot of waste in the form of rework and added complexity.

This goes beyond simply bugs in code. Since Lean looks across the entire development process, all of the following symptoms indicate problems with quality:

- Building something with insufficient quality (bugs)
- Building something the customer did not ask for because a requirement was misunderstood
- Building something the customer did ask for, but later realized it was not what they meant and they do not want it now
- Building something the customer described properly but once they saw it realized that they had asked for the wrong thing
- Building something the customer did ask for and it was what they meant but now they want something different

From the customer's point of view, all of these are defects in the product. Whether a bug, a lack, or an excess, they do not consider these valuable. And building anything that the customer does not find valuable is waste. We want to reduce or eliminate waste and so improve quality. The responsibility for eliminating waste lies with the team.

Removing Bugs Is Waste.
Lean would say that there is no value in taking bugs out of systems. Why? One reason is that putting a bug into code and then taking it out adds no value. No customer has ever asked any development team to do this. Ever! Let's make this personal to drive home the point: Suppose you take your car in to the dealership for a $50 oil change. When you pick up your car, the bill is $550. When you complain, "The oil change was supposed to be only $50!" they respond, "Well, it was $50, but we charged you $500 to take the dent out of your fender." "But there was no dent in the fender when I brought it in!" "True, but we dented it when we changed your oil!" Did you receive value from their taking the dent out?

Where to start? Lean thinking says to look at the system within which a defect arises rather than seeking someone to blame. In other words, look to systemic causes rather than to problems with individual performance. In software development, one typical systemic problem is the separation of the development team from the test team. Developers create code and the test team tries to detect errors and help fix the code. Yes, the testing team is addressing the defect problems, but the system as a whole allows the defects to arise in the first place.

The practice of testing bugs (defects) out of a system is akin to Deming's famous quip, "Let's make toast American style. You burn, I'll scrape!" How can this be improved? Can we reduce the occurrence of

defects? Wouldn't that be a better use for quality assurance rather than having them try to intercept defects after the fact?

Yes! In Lean-Agile,

The primary role of QA is not to find defects but to prevent them.

Quality assurance provides an opportunity to discover why bugs occur. QA's role should be to improve the process so as to prevent defects and to use found defects to improve the process from which they sprang. In other words, when errors occur, take that as an indication that something about the process could be improved—and find an improvement that will prevent the error from occurring again.

QA at the End of the Cycle Is Inherently Wasteful

Placing quality assurance at the end of the development cycle contributes significant waste to the process. Let's see why.

Consider the typical development process where QA is done at the end of the cycle.

1. The analyst and the developer discuss the requirement to be implemented.
2. The developer goes off to develop it and
    a. Writes some code,
    b. Considers how she will know she has implemented it properly,
    c. Runs her own tests to confirm it is correct (often, this is done by hand), and then
    d. Hands off her code to QA for testing.
3. QA discusses the requirement with the analyst and determines what a good test case would be.
4. QA implements this test.
5. If (when) an error is found,
    a. QA tells the developer there is a bug,
    b. The developer investigates by looking at QA's test case, and then may
    - Discover her bug and go on to fix it
    - Discover her bug and put it in a queue
    - Believe she has the correct test case and that the QA person has done the wrong thing and then get into a dispute over how to resolve the error

At best, this approach has a lot of redundancy built in. In addition, there are numerous opportunities for miscommunication. And when an error occurs, there will likely be a significant delay until it is discovered.

Improve Results by Moving QA Up-Front

Moving QA closer to the front of the value stream can reduce or eliminate redundancy, miscommunication, and delay. One way to do this is to ensure that whenever a requirement is stated, the team asks the question, "How will I know I've done that?" The answer should be in the form of specific inputs and outputs. Because they are specific examples they can be considered to be a test. These answers need either to come from the customer or be validated by her. The answers should be in the form of an acceptance test of the code produced.

Getting this question asked and answered before a developer starts working has several advantages. First, it avoids the redundancy of asking and answering multiple times by different people. Second, it gives the developer something to shoot for: guidance to get the job done. Finally, if the test is automated, it can be run any time the code is altered and thereby verify that the code still works—or give warning that it does not. It creates a context for a better and more reliable conversation between developer and analyst. [1]

Let's consider an example. Suppose you are given the following requirement:

Basic Employee Compensation. *For each week, hourly employees are paid a standard wage per hour for the first 40 hours worked, one-and-a-half times their wage for each hour after the first 40 hours, and two times their wage for each hour worked on Sundays and holidays.*

Since we have moved QA up front, the team starts by asking, "And how will I know I have done that?" They might come up with the tests shown in Table 9.1.

Table 9.1. Examples Describe Tests for the Requirements

StandardHours	HolidayHours	Wage	Pay()
40	0	20	$800
45	0	20	$950
48	8	20	$1520

The developer starts reviewing these tests: Looking at the first row, he says, "OK, I see this; 40 hours times $20/hour gives me $800. That makes sense. But 45 hours times $20 should give me $900, not $950. Oh, but I see, I forgot the time-and-a-half, so it's really 40 times $20 plus 5 times $30 so $950 is correct. But let's see, 48 standard hours means 40 times $20 plus 8 times $30 plus the holiday hours at double-time

Misunderstanding Is More Natural than Understanding

It is always dangerous to assume that what you heard is what was said. Communication requires work and is built on common understanding between those communicating. English is an ambiguous language. Many words are actually their own antonyms. For example, "clip" can mean "to put together" (as in "clip the coupon to the paper") or "to take away" (as in "clip the coupon from the paper"). Thus, "clip the coupon" is ambiguous.
However, trying to avoid this ambiguity by writing everything down can lead to something enormous, like the requirements document for a tank Winston Churchill once described by saying, "This paper, by its very length, defends itself against the chance of being read."

[1] I got this insight by reading Rick Mugridge's fabulous book *FIT For Developing Software* (Mugridge 2005). The beginning of this book is a must read even if you are not going to use FIT.

should give me $800 + $240 + $320 or $1360. Not $1520."

This is confusing! So the developer talks to the customer/analyst and the tester to clear it up. One of them explains, "Well, the 48 standard hours gives you $1040 but the 8 holiday hours, since they are overtime, are paid at time-and-a-half of double-time or $480 for a total of $1520." Ah! Things become clearer!

Notice how having the test specified with the requirements changes the conversation between the developer and the analyst? Moving the testing process to the start of the creation process—before coding—makes it more likely that the developer will build what the customer intends.2

In the previous example, the team gets the benefit whether or not the tests are automated. They benefit simply by better communication. While some arguments can be offered against automating testing, it is hard to argue against at least specifying the tests up front. It adds no extra work and it creates great value. This should always be done.

The first task in implementing any story should be to answer the question, "How will I know I've done that?" Often, this must done in the iteration before the team does the story (or in the planning on the day of the iteration in which they intend to do the story), since it may take considerable time as well as be required in order to size the story.[3]

When the Product Champion Will Not Answer Your Questions

Specifying tests up front is powerful, as long as you can do it. What if the analyst, Product Champion, or customer representative cannot or will not answer the question, "How will I know I've done that?" What do you do then? In our classes and consulting, we have heard so many teams complain that their "customer" (meaning whomever is representing the customer) just wants the team to figure things out for them. The customer does not want to help, does not want to be bothered, or simply wants to leave it to the "professionals" who probably know better how to do things in software projects (right?). Should the developers just go ahead and do the work for the customer? No! Help the customer, but don't do it all for them.

The team can develop preliminary answers to the question, "How will I know I've done that?" and then take the tests to the customer to ask, "If we do this, will we have met your needs?" Giving them something concrete to work with makes it easier for the customer to answer the question. In virtually every case where we have seen teams take this approach, the customer makes the time to get the specifications completed. And if they don't, you probably should not proceed.

As Jeff Sutherland says (Sutherland, 2003),

[2] In our classes, before we specify the test we ask people to name all of the different possible misunderstandings of the stated requirement. Usually another half dozen not mentioned here are stated. When I ask "how do you know you've got them all" people realize it is impossible to know what you don't know.

[3] This may seem contrary to some Agile concepts; however, in real-world large projects, standard Scrum practices are often overly simplistic.

"The task then is to refine the code base to better meet customer need. If that is not clear, the programmers should not write a line of code. Every line of code costs money to write and more money to support. It is better for the developers to be surfing than writing code that won't be needed. If they write code that ultimately is not used, I will be paying for that code for the life of the system, which is typically longer than my professional life. If they went surfing, they would have fun, and I would have a less expensive system and fewer headaches to maintain."

If you cannot get verification of what is needed, you will almost certainly build something that is not useful. And, even worse, you will be adding complexity to your code base. That is waste. And waste is an enemy.

If the customer cannot or will not confirm that you have delivered what they want, you should simply state that you believe that the customer does not value the feature; that it is not a high priority. If it were valuable, they would make the effort to specify the tests. Moreover, you should tell your management team that you recommend *not* building the functionality. If you are required to build it anyway, go ahead, but know that it could well turn out to be a waste of time.

There is an exercise at the end of this chapter that you should do with another team member. It helps make apparent that when the "customer" won't help clarify their needs, the functionality that is delivered is almost always not what was needed.

Executable Specifications and Magic Documentation

Tests represent documentation for the requirement. They describe very specifically how the system behaves. Even better, when tests are made this explicit, they can be put into a testing framework, such as Framework for Integrated Test (FIT), and these tests can actually be written by non-developers. This makes it relatively easy to tie the tests to the code. At any time, the tests can be run to ensure that the code is giving the desired results.

In the previous example, a test run might produce results such as those shown in Table 9.2. The rows for which the tests have run successfully are green while those that have failed are red (with both the expected and actual results shown). If the tests are not connected to the code they would be shown in yellow.

Table 9.2. Example of FIT Tests When Executed

StandardHours	HolidayHours	Wage	Pay()	Result (Color)
40	0	20	$800	**Green**
45	0	20	$950	**Green**
48	8	20	*$1360*	**Red** (expected $1520 but got $1360)

This is powerful. Such automation provides for "magic" documentation. Most documentation just sits there when the system doesn't correspond to what is stated. But automated test specifications represent executable specifications and turn red (magic!) when they break. Besides the improvement to

documenting requirements, these automated tests provide a safety net when the code is changed and allow for shorter iterations with lower costs for the accompanying regression testing.

Acceptance Test-Driven Development

Test-Driven Development (TDD) began as a coding method based on writing unit tests for functions prior to writing the functions themselves. The main drivers behind TDD were to 1) ensure an understanding of what the functions were to do, 2) verify that the code was doing it, and 3) take advantage of the fact that this kind of design process (defining tests before writing code) improved the quality of the code written. Classic TDD, however, is primarily a developer function. It is about reducing technical risk—the risk that the software implementation won't match what is being asked for.

Once we bring together the customer, quality assurance, and developer roles, a new perspective on test-driven development emerges: the concept that test-driven development should seek to reduce both technical *and* market risk. Reducing market risk means both identifying properly the needs of the marketplace and accurately conveying them to the development team. Reducing technical risk requires ensuring that the code works as it is designed to.

By driving code development from acceptance tests, both of these risks should be reduced. The conversations described previously in this chapter reduce market risk because they assist the customer, tester, and developer in understanding one another. Once acceptance tests have been defined, creating smaller tests that implement aspects of the acceptance ensure that the code works properly. This is called Acceptance Test-Driven Development (ATDD).

Perspective Can Make All of the Difference. To best understand the power of perspective, consider this conversation we once had with someone who was considering Agile methods for his company that built instruments for the health care industry. He asked how he could verify that he had complete test coverage if he built up his test cases on small stories. Since he knew that we were Agile consultants, he expected some fancy answer that would alleviate his concern. But instead we told him that you couldn't—at least not easily and not with great certainty. We said that instead of creating stories and putting them together—and therefore starting with test cases and putting them together—you need to start with acceptance tests and break the stories down.

Perspective again is a big contributor to effectiveness here. Classic TDD is often about writing unit tests for functions. It is not uncommon to see acceptance written as a combination of unit tests. But we're strongly suggesting that going the other way makes much more sense—writing unit tests based on acceptance tests. This extends TDD into a process of ensuring an understanding that the tests are complete. In other words, start at the top (acceptance) tests and break them down instead of starting at the bottom (unit) tests and putting them together.

This top-down approach can be used to ensure test coverage of the needed behavior is complete. This is not unlike starting with a jigsaw puzzle that is put together—to make sure you have all of the pieces—

and then breaking it up so you can put it together. If you start with all of the pieces separated from each other, you can't be sure you have it all right until you've put the entire puzzle together.

Summary

Quality Assurance should be responsible for *preventing* defects, not merely for finding them. To achieve this, QA should be moved up to the front of the development cycle. This helps teams avoid many of the communication errors that often result in delays, defects, and waste. Before each story is implemented, the team and the customer should ask and answer the question, "How will I know I've done that?" If possible, tests should be implemented before code is written, thereby assisting the developers in both seeing how their code behaves and ensuring that high-quality code is produced with minimal waste.

Try This

This exercise is best done as a conversation with someone in your organization. After the exercise, ask each other if there are any actions either of you can take to improve your situation.

The purpose of this exercise is to help you and your group understand the value of asking the question, "How will I know I've done that?"

Do this exercise with someone else if you can.

1. Consider a time when you developed code, only to discover later from the customer that what was developed was not what they meant when they asked for it. Make sure this was a time you did not asked the question, "How will I know that I've done that?"
2. How would the result have been different if you had asked that question?

An anecdote about this exercise: We run this exercise in all of our Lean and/or Agile/Scrum classes. In one class, the dialogue was particularly animated. People discussed how they had tried to get this question answered but could not get the product manager (their customer rep) to answer it. This was mostly because the product manager didn't know himself. They went ahead and built things as best they could but later ended up essentially having to redo the system. We had the group reflect on all the times that this had occurred in the last couple of years for any of the projects that the teams had worked on. They realized that *every* time this had occurred they had had to redo the code. Not only had they wasted much effort, they had also made their system much more complex than it should have been. They resolved *never* to write code unless they could get the question "How will I know I've done that?" answered. I've often said there are not any best practices that all teams should follow, but if there is one – it is this!

Recommended Reading

The following works offer helpful insights into the topics of this chapter.

Mugridge, Rick. *Fit for Developing Software: Framework for Integrated Tests.* Upper Saddle River, NJ: Prentice Hall PTR, 2005.

Sutherland, Jeff. "Get Your Requirements Straight." *Jeff Sutherland.* March 11, 2003. http://jeffsutherland.com/scrum/2003/03/scrum-get-your-requirements-straight.html (accessed March 13, 2009).

Business-Driven Software Development™

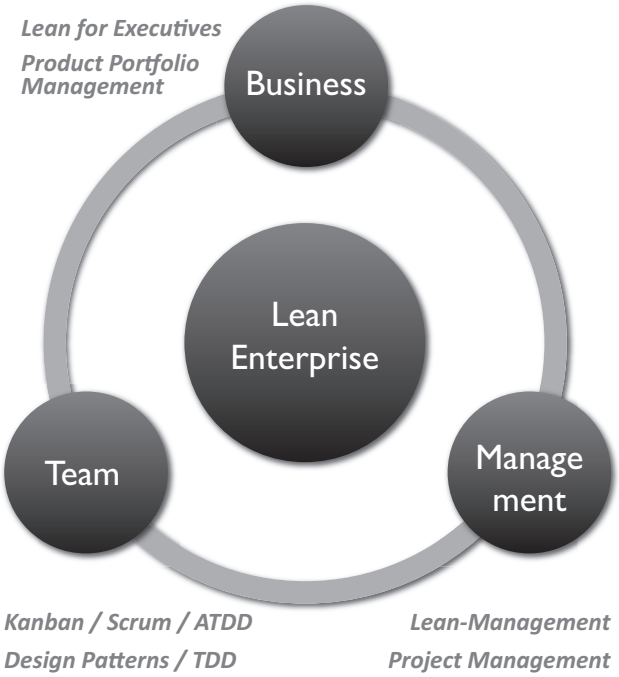
Assessments, Consulting, Training, and Coaching

Business-Driven Software Development (BDSD) is Net Objectives' proprietary integration of Lean-Thinking with Agile methods across the business, management and development teams to maximize the value delivered from a software development organization. BDSD has built a reputation and track record of delivering higher quality products faster and with lower cost than other methods

BDSD goes beyond the first generation of Agile methods such as Scrum and XP by viewing the entire value stream of development. Lean-Thinking enables product portfolio management, release planning and critical metrics to create a top-down vision while still promoting a bottom-up implementation.

BDSD integrates business, management and teams. Popular Agile methods, such as Scrum, tend to isolate teams from the business side and seem to have forgotten management's role altogether. These are critical aspects of all successful organizations. In BDSD:

- **Business** provides the vision and direction; properly selecting, sizing and prioritizing those products and enhancements that will maximize your investment
- **Teams** self-organize and do the work; consistently delivering value quickly while reducing the risk of developing what is not needed
- **Management** bridges the two; providing the right environment for successful development by creating an organizational structure that removes impediments to the production of value. This increases productivity, lowers cost and improves quality



Become a Lean-Agile Enterprise

All levels of your organization will experience impacts and require change management. We help prepare executive, mid-management and the front-line with the competencies required to successfully change the culture to a Lean-Agile enterprise.

Prioritization is only half the problem. Learn how to both prioritize and size your initiatives to enable your teams to implement them quickly.

Learn to come from business need not just system capability. There is a disconnect between the business side and development side in many organizations. Learn how BDSD can bridge this gap by providing the practices for managing the flow of work.

Why Net Objectives

While many organizations are having success with Agile methods, many more are not. Much of this is due to organizations either starting in the wrong place (e.g., the team when that is not the main problem) or using the wrong method (e.g., Scrum, just because it is popular). Net Objectives is experienced in all of the Agile team methods (Scrum, XP, Kanban, Scrumban) and integrates business, management and teams. This lets us help you select the right method for you.

Assessments

See where you are, where you want to go, and how to get there.

Business and Management Training

Lean Software Development

Product Portfolio Management

Enterprise Release Planning

Productive Lean-Agile Team Training

Team training in Kanban, Scrum

Technical Training in ATDD, TDD, Design Patterns

Roles Training

Lean-Agile Project Manager

Product Owner

For more information about Business Driven Software Development Services
Contact Mike Shalloway at Net Objectives: mike.shalloway@netobjectives.com / 404-593-8375
NetObjectives.com / 1-888-LEAN-244 (888-532-6244)