Chapter 6

QUESTIONING QUALITY

It's no secret that compromised quality was a regular feature of traditional waterfall software projects, primarily due to the highly risky practice of all-in-one integration and testing at the very end. Although Scrum's focus on iterative and incremental development alleviates significant risk, the reality is that we will never be able to totally eliminate every defect.

The following three shortcuts step you through both preemptive measures and remedial actions to help deal with those pesky bugs.

Shortcut 16: Bah! Scrum Bug! lays out a selection of new definitions, principles, and processes to assist in managing defects during sprints. Shortcut 17: We Still Love the Testers! considers the new roles that a tester now plays on a Scrum team. Finally, Shortcut 18: Automation Nation identifies a selection of important starting points when commencing down the path of test automation.

Shortcut 16: Bah! Scrum Bug!

Although we no longer have to contend with actual moths infiltrating our vacuum tubes (yep, that's where the term *bug* originates), their digital descendants are still regular visitors to every codebase on the planet. In the same way that bugs have changed over time, so has the way that they are dealt with, and this is particularly pertinent to our new agile way of thinking.

Previously, our waterfalling world viewed the handling of bugs very sequentially (see Figure 6.1).

With this simplified process as the benchmark, let's look at what needs to change when implementing Scrum. In particular, it is important to appreciate that programming and testing need to be conducted in tandem rather than in sequential phases if teams hope to deliver working functionality early and often.

New Definitions

Before exploring some new Scrum-friendly processes for handling bugs, I'd like to set up some foundations with a few definitions and principles that I like to use.

Definition 1: Issues

- An *issue* is a problem that occurs during the sprint and is tightly coupled to a user story (see Shortcut 10) that has not yet met its definition of done (see

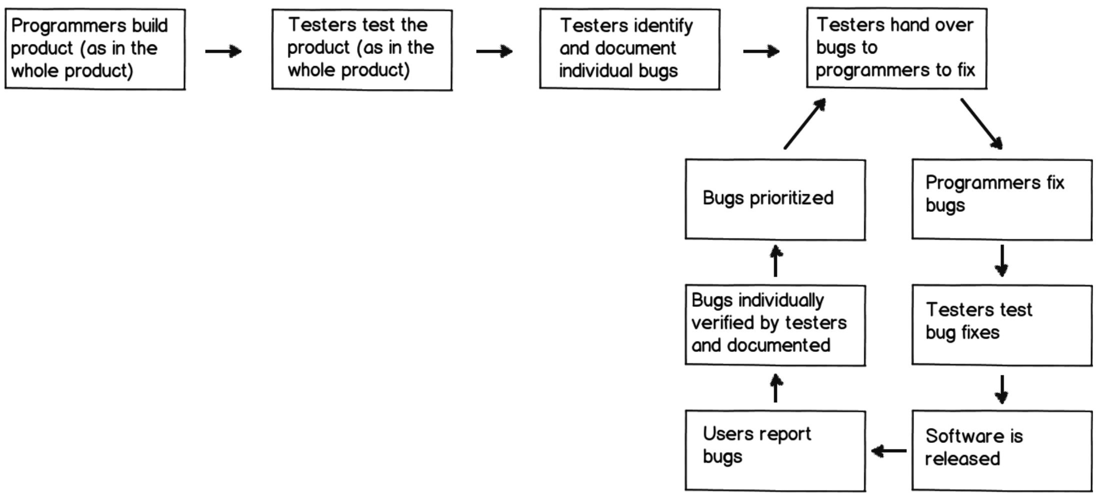
Programmers build product (as in the whole product)
Testers test the product (as in the whole product)
Testers identify and document individual bugs
Testers hand over bugs to programmers to fix
Programmers fix bugs
Testers test bug fixes
Software is released
Users report bugs
Bugs individually verified by testers and documented
Bugs prioritized

FIGURE 6.1 A simplified flow of how bugs were sequentially handled in the waterfall world.

Shortcut 11). Issues will therefore typically be picked up during the sprint (that the corresponding user story is being worked on) either by the programmer, the automated build (see Shortcut 18), a tester conducting exploratory testing, or the product owner during a walkthrough (see Shortcut 12).

- An issue is *not* a product backlog item (PBI). Instead, an issue should be seen as part of the evolving acceptance criteria for a user story. Essentially, what I am saying is that until the issue is resolved, the user story is not complete, and that being the case, an issue is a part of the actual user story rather than an independent, albeit associated, product backlog item.

Definition 2: Bugs

- A *bug* is a bug only if it is identified *after* a user story has been completed and accepted by the product owner. Bugs will therefore typically be picked up by users (postrelease) or via an automated regression test (following the implementation of subsequent user stories).
- A bug is a type of product backlog item. A user story is another type of product backlog item. Bugs and user stories should be prioritized together in the same product backlog and estimated using the same approach, such as relative estimation (see Shortcut 13). A particular bug may relate to a specific user story, but it should be treated independently as far as any tracking and prioritizing is concerned. To reiterate a point made in Shortcut 10, a bug can theoretically be represented utilizing the user story format, although I personally don't find it to be a suitable format in most cases.

New Principles

Now let's look at the three new principles.

Principle 1: Remove the Red Tape

Recall the second principle from the Agile Manifesto: "Working software over comprehensive documentation" (Beck et al. 2001). During my earlier years getting "soaked" in the waterfall world, I observed that a nontrivial amount of time was spent by both testers and programmers carefully documenting in painstaking detail the relevant bug minutiae. I remember regularly asking myself whether it was actually taking longer to document or to fix the damn bugs. Scrum relies on as much real-time communication as possible (rather than formalized, written bug reports), but if documentation is required, it should be fit for purpose and kept to the bare minimum.

Principle 2: Address Issues Immediately

There is nothing worse than your own stale code. Oh, wait a minute, yes there is—someone else's stale code! Sadly, back in the day when we all used to follow the sequence detailed at the beginning of the shortcut, it was very common to have to return to work on bugs in code that we had well and truly moved on from. The spin-up time to get back into the headspace to address old issues (be they yours or those of a colleague who's off on vacation) is significant and, frankly, a big fat waste of time. The sooner an issue is found, the cheaper it is to fix, and that is why with Scrum, testing is completely entwined with the programming.

Principle 3: It Ain't Over 'Til It's Over

Bottom line is that unless a user story meets the definition of done (see Shortcut 11), it might as well not exist as far as the customer is concerned. Customers are simply interested in final results and obtaining business value. If a user story is not yet done, it needs to be the top priority for the developer(s) working on it, and they should avoid moving on to any new work until it is completely done and dusted!

New Approaches

With our new bug-handling foundations now established thanks to the previous definitions and principles, let's focus on some approaches that I recommend you follow within the sprint:

- **Scenario 1: A tester is conducting some final exploratory testing on a user story and discovers an issue.**

  First, because the user story should be the top priority of the programmer working on it (see Principle 3), the tester should feel free to walk over to the programmer and explain and/or demonstrate the issue as soon as it is found. Again, because the user story is the top priority, the programmer should drop whatever he or she is doing and immediately jump on the issue. In this situation, there isn't any requirement for written documentation, as the verbal discussion will suffice, assuming that the issue is immediately resolved and verified.

- **Scenario 2: Same as Scenario 1, but this time, the programmer in question is already in the middle of resolving a different issue (related to the same user story).**

  In this case, the tester, after finding another issue, looks over and sees the programmer with headphones firmly in place and in the zone fixing a previous issue. The last thing the tester wants to do is to disturb the programmer mid-fix. As such, it is important to capture the details somewhere so that the tester can continue with the exploratory testing without forgetting the details of the new issue.

  As discussed in the definition of *issue*, an issue should be treated as part of the acceptance criteria of the user story, saving the tester the pain of creating a new bug, classifying it, assigning it, prioritizing it, and so on. Instead, my recommendation is that the tester simply add a line item to the acceptance criteria with a date/time reference, initial it, and add some bullet point details of the issue. When the programmer is free, a discussion can take place using the notes as a prompt. Also, the documentation ensures that the programmer can get on with the resolution even if the tester is not around for whatever reason.

- **Scenario 3: During the final user acceptance testing for a release, a range of trivial user interface bugs are found that were somehow missed during development.**

  Again, let's try to reduce the time spent on unnecessary administration; in this situation, I recommend that a single PBI be created as a container for the collection of minor bugs. Each specific fix may take only minutes, so creating individual PBIs for each issue could end up taking longer than the actual fix-ups!

  I recommend following this approach only if

  – The trivial bugs are of a similar priority level.

  – They are somewhat related and it makes sense to tackle them at the same time.

  If these conditions aren't met, then simply create separate PBIs for the items even if they are seemingly trivial in their own right.

- **Scenario 4: During a sprint, a critical bug is found in production requiring some of the Scrum team to resolve it.**

  The first question to ask is, *How critical is critical, or more specifically, can it wait until the next sprint?* As detailed in Shortcut 1, the last thing you want to do is to change the goal of the sprint. Assuming that the production bug can wait, it should be captured as a PBI, entered into the product backlog, prioritized by the product owner, and tackled potentially in the next sprint planning session.

  However, what should happen if the discovered product bug is one of those dreaded villains that simply can't wait? Well, we then need to ask another question: *How long will it take to fix the bug?* If you recall from Shortcut 8, it isn't wise to max out team capacity for working exclusively on new sprint tasks to provide some room for handling non-project-related disruptions. As such, this buffer time can also be allocated to resolve the occasional emergency bug without disrupting the sprint.

  If, however, the resolution will take longer than the buffer time, you have two choices. First, you can treat these issues as impediments and track them accordingly (see Shortcut 9), or, if the issue is such a major drama (to the extent that it completely destroys the sprint goal), there is always the undesirable fallback position: a sprint cancellation that can be called by the product owner. A cancellation will end the current sprint and send the team back to sprint planning.

Turning Moths into Butterflies

Bugs can certainly cause pain, and like it or not, they're never going to become an extinct species. However, what we have learned are better ways to deal with them. We now know that disposing of fresh bugs is easier than having to deal with old, festering ones and that spending unnecessary time documenting every issue is a waste of time.

Scrum handles testing and bugs very differently from traditional approaches. By adopting these new definitions and principles, you will start to avoid the unnecessary overhead and communication breakdowns that have previously stopped teams from turning those moths into less ugly butterflies.

Shortcut 17: We Still Love the Testers!

In fact, not only do we still love the testers, we love them even more in our new Scrum world! I really feel that this is an important point to emphasize, and I'll tell you why.

I remember when I was excitedly presenting Scrum 101 concepts to my first soon-to-be Scrum team. I was sure everyone was going to pick up on my infectious enthusiasm, and indeed I gleaned a whole bunch of decisive nods and smiles. However, when I looked more closely, I started to also observe some noticeable fidgeting and darting eyes (synonymous with discomfort and fear) among a few of the testers. To understand this discomfort, we need to look into the past and briefly explore what has happened to the testing function in recent times.

Waterfall Friendship

Thanks in large part to the earlier adoption of the traditional waterfall model, a more profound appreciation for the testing function began to take hold. Within many organizations, a strong, independent testing team that stood on more of an equal footing with the programming team was becoming the norm. Testing standards were developed, professional development paths purely for testers were established, and the test team "owned" a whole phase of the cascading waterfall process.

Then along comes Scrum (and its other agile cousins), and all of a sudden, life changes. Testing becomes the responsibility of everyone on the team, unit testing becomes a programmer-centric practice, and even functional tests can be automated by programmers. Suddenly the question starts creeping into worried minds: *How and where do the testers fit in?* Before going on, and to avoid any undue panic for readers at this stage, I will cut straight to the chase and state that the tester has never been so important. Lisa Crispin and Janet Gregory (2009), authors of *Agile Testing*, emphasize that the whole-team approach is one of the biggest differences between agile development and traditional development. Some testers recognize this difference and are immediately relieved and excited by Scrum, and others remain fearful of the new world order.

Change Is in the Air

Change is scary. Crispin and Gregory offer some important insight into why the transition to agile development can be particularly worrisome for some testers. They contend that "loss of identity fear" is at the heart of a tester's concerns, and following is a selection of these specific fears:

- Fear that they will lose their QA identity
- Fear that they lack the skills to work in an agile team and will lose their jobs
- Fear that when they're dispersed into development teams, they won't get the support they need (Crispin and Gregory 2009)

As I will explain, when working in a genuine Scrum environment, none of these fears are justified. Yes, there is an identity shift; however, all of the worthy testers I have worked with have either immediately or eventually embraced their enhanced identity with open arms.

When change occurs, it is a natural instinct to romanticize the past, clinging to anything that was warm and fuzzy rather than remembering the darker, negative times. Testers shouldn't forget that life certainly wasn't a walk in the park in the old days (even if there were pretty waterfalls along the way). An image that I have etched into my memory is that of the frazzled, worn-out testers at the end of a waterfall project. "Traditional test teams are accustomed to fast and furious testing at the end of a project . . . in agile projects, you are encouraged to work at a sustainable pace" (Crispin and Gregory 2009).

New Identities

How do we help testers embrace their role in the new world where the waterfalls have dried up?

Let's first address the elephant in the room: the fear of being made functionally redundant. Fundamentally, the testers should feel safe because they are different. They possess a unique skill set and a way of thinking that is critical to the success of any software project. I like to use the description offered by Nick Jenkins in "A Software Testing Primer" (2008) to help illustrate this point:

> There is a particular philosophy that accompanies "good testing." A professional tester approaches a product with the attitude that the product is already broken —it has defects and it is their job to discover them. . . . Developers approach software with an optimism based on the assumption that the changes they make are the correct solution to a particular problem. . . . By taking a skeptical approach, the tester offers a balance. They seek to illuminate the darker part of the projects with the light of inquiry.

In a nutshell, testers think in alternative problem-solving patterns that are, generally speaking, mutually exclusive to the way programmers think.

Now that we've got that big concern out of the way, let's explore some of the exciting new subidentities that a Scrum tester can assume that are clearly above and beyond the mind-numbing, repetitive manual testing that previously played such a disproportionately large part in a tester's life (see Figure 6.2).

The Tester as a Consultant

Testers are specialists at their craft, and as such, they are in a unique position to help guide nontesters to improve their testing game. With Scrum's focus on delivering quality working software on a regular basis, this has never been so important. For starters, the tester can (and should) act as a sounding board for the programmers as they start to get their heads around test-driven development.

Also, while pair programming is certainly a powerful Extreme Programming (XP) technique (that is sometimes adopted by Scrum teams), I feel that "pair testing"

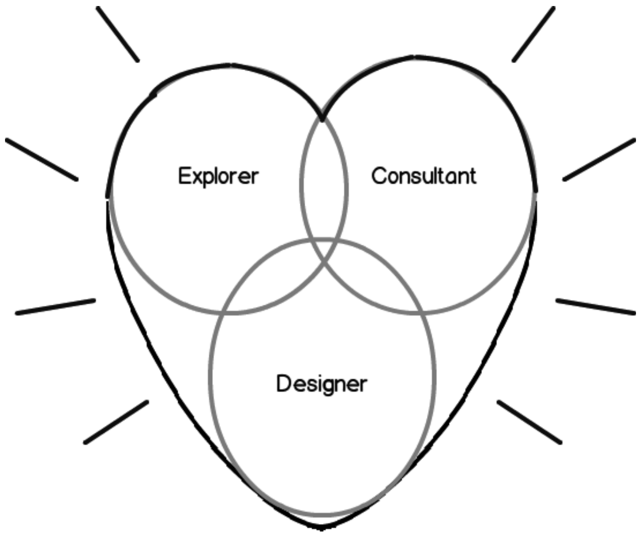

FIGURE 6.2 In Scrum, we still love the testers, especially with their new Scrum identities.

(when a tester pairs up with a programmer) is potentially even more powerful because there is additional scope to encourage functional skills transfer. It also fosters further appreciation for one another's skills and abilities.

Consulting to the user-experience designers can also be of significant benefit by helping to anticipate potential issues associated with the more complex workflows.

Finally, the product owner can no doubt leverage the tester's inherent understanding of the core acceptance criteria by assisting in various intra-sprint walkthroughs (see Shortcut 12) and helping with the final verification of the done user stories.

The Tester as a Designer

I believe that the core skill of a tester is actually that of design. Irrespective of who actually runs or implements a test, a seasoned professional tester will always be able to design the most effective test cases compared to anyone else on the team.

Well-designed tests not only form the foundation for the eventual testing itself but can also provide vital input into the technical design that takes place during sprint planning. When a tester is involved in the design of a user story's test cases prior to the sprint planning session, I can assure you that the meeting will be a great deal smoother and faster with fewer contentious debates. For those concerned that this advice is slipping into the realm of waterfalling sprints, I support Mike Cohn's (2009) thoughts:

> Being part of the team on this (current) sprint and spending some time looking ahead is not the same as working a sprint ahead of the team. . . . their top priority is delivering whatever is committed for the current sprint. Beyond that, their job is to look ahead in exactly the same way everyone expects a product owner to be looking ahead.

The Tester as an Explorer

As you will read in Shortcut 18, test automation is integral to the success of Scrum. However, even with extremely thorough test automation in place, there will always be the need for manual exploratory testing that no level of automation is able to replicate. This element of testing is without doubt more art than science, and for those under the false impression that exploratory testing is just another name for gorilla or ad hoc testing, the following commentary by Crispin and Gregory (2009) will give you a new appreciation for the subtlety of this function:

> 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.

A New Beginning

In a Scrum team, everyone is responsible for testing. Quality is no longer an afterthought, and testing should become an inherent part of every stage of the user story development, including before a single line of functional code is written.

The transition to Scrum should feel like an exciting rebirth for the tester. Removing the manual testing shackles offers Scrum testers an opportunity to focus on what they do best: design, consulting, and exploratory testing. They are finally given an opportunity to flex their unique skill set in far more interesting ways than before.

Shortcut 18: Automation Nation

You’ve got a simple choice: jump on the automation bandwagon, destined for exciting Scrum-filled destinations or suffer a trip down the slippery slopes of “Scrummer-fall.”

Trying to implement Scrum without automation is like trying to drive a sports car on a beaten-up dirt track—you won't experience the full potential of your exciting vehicle, you will get horribly frustrated, and no doubt you will end up damaging and probably blaming the car. As James Shore and Shane Warden (2007) point out,

> Software development is demanding. It requires perfection, consistently, for months and years of effort. At best, mistakes lead to code that won't compile. At worst, they lead to bugs that lie in wait and pounce at the moment that does the most damage.

The Scrum Guide, written by Ken Schwaber and Jeff Sutherland, doesn't make mention of software engineering practices at all; in fact, the words *software* and *engineering* do not appear once in the guide. Rather, Scrum is abstracted above this layer, described more generically as a "framework for developing and sustaining complex products" (Schwaber and Sutherland 2011).

That being said, you won't hear one genuine expert say that Scrum (as it applies in the software context) doesn't work significantly better when it is combined with strong, automated software engineering practices such as those that you will find in the Extreme Programming (XP) set of practices (Beck 1999). This shortcut explains why.

Automation is a massive topic—numerous books are dedicated to it exclusively. This shortcut simply gives you some general advice to start you on your automation journey. There are many layers, many tools, and various combinations of tools; however, I intend to keep this shortcut nice and straightforward to avoid your getting analysis paralysis.

We focus on several key automation practices that are all heavily intertwined, including continuous integration, test automation, build/deploy automation, and the relatively new concept of continuous delivery.

Continuous Integration (CI)

Martin Fowler (2006), one of the original signatories to the Agile Manifesto, describes the core practice of continuous integration:

> Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily—leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.

What's the key benefit, then (if it's not already obvious)? Again, I don't think I can describe it any better than Fowler does:

> Integration is a long and unpredictable process . . . . The trouble with deferred integration is that it's very hard to predict how long it will take to do, and worse, it's very hard to see how far you are through the process. The result is that you

are putting yourself into a complete blind spot right at one of the [most tense] parts of a project—even if you're one of the rare cases where you aren't already late.

Getting CI up and running is certainly a great place to start your automation journey. Scrum trainer, Kane Mar (2012) further emphasizes this point: "Developing an increment of potentially shippable code without CI is almost impossible after the 3rd or 4th sprint simply because the amount of change and regression testing becomes overwhelming." CI is a classic embodiment of the proverb, "A stitch in time saves nine." By frequently ironing out any small integration issues on a day-to-day basis rather than battling a vast array of compounding integration problems at the end of a project, teams save themselves from considerable stress and anguish.

The CI server constantly monitors for any new code that, when checked in, automatically triggers a new build. In the process, the server can (and should) be configured to run any automated tests. Because numerous CI builds will be run every day, it is critical to ensure that the build is very quick—any longer than 10 to 15 minutes will cause bottlenecks in development and defeat the whole purpose. This 10- to 15-minute time constraint could mean that the CI build won't include all of the slower functional tests that instead get run during the secondary build (we cover this topic later in the shortcut).

Test Automation

It doesn't take a genius to work out what will happen if test automation isn't introduced. By about sprint 3 or 4, the amount of manual regression testing will become so significant that some team members may feel tempted to resort to running a dedicated " testing sprint" (see Figure 6.3). This is not a good idea, but even worse, some

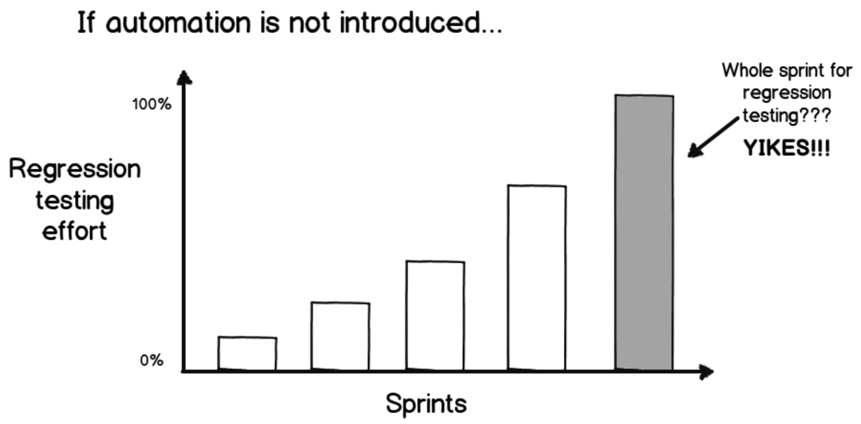

FIGURE 6.3 If test automation is not introduced, a "testing sprint" could be just around the corner.

bright spark may even suggest that a "testing team" should work a sprint (or more) behind to catch up on the regression testing. If either of these situations occur, your team has begun the sad decline back into the nonincremental world of traditional waterfall development.

The good news is that with test automation, there is no reason to fear slipping back to the dark ages. The bad news, as Crispin and Gregory (2009) explain, is that "automation requires a big investment, one that may not pay off right away. It takes time and research to decide on what test frameworks to use and whether to build them in-house or use externally produced tools."

While I agree that an investment is needed, I am also a firm believer that the positives far outweigh the negatives. Some of the benefits provided by Crispin and Gregory include the following:

- Manual testing takes too long.
- Manual processes are error prone.
- Automation frees people to do their best work.
- Automated regression tests provide a safety net.
- Automated tests give feedback early and often.
- Tests and examples that drive coding can do more.
- Tests provide documentation.
- Automation can be a good return on investment.

Types of Automated Testing

It can be a little daunting when you are first introduced to the vast world of test automation. Not only are there numerous test layers, but the other complication is that there is still a lack of industry-level standardization around the naming conventions as well as the exact scope of each layer. As such, the descriptions and naming of the various layers that I use next may vary slightly from those you might be familiar with.

Unit Testing

Unit tests focus on testing the lowest-level, independent programming blocks (such as a method in a class) and are usually implemented with one of the xUnit frameworks. These tests should be implemented via test-driven development (TDD), which provides an additional spinoff benefit:

> If your programmers are using TDD as a mechanism to write their tests, then they are not only creating a great regression suite, but they are using them to design high-quality, robust code (Crispin and Gregory 2009).

For those of you who are unfamiliar with TDD, Ron Jeffries (2010), Agile Manifesto signatory and Scrum trainer, offers a great explanation:

> Test-Driven Development requires me to write all the code, and only the code, that is needed to pass tests, which I write first. This discipline helps me focus on what the code must do before I focus on how it would do it, and it results in code which is simple and quite testable. TDD is not a rote, stupid practice. It is, instead, an almost meditative way of keeping my mind focused on what's going on. It reduces my defects a lot, and my tension even more.

Functional Testing

Functional testing is also often called *acceptance testing*. Perhaps one day we will call it *user story testing*, as the idea is to be able to test and automate the full end-to-end functionality of a particular user story.

These types of tests may or may not include automating the testing of the user interface (UI). This additional layer typically incurs additional cost because of the time it takes to run tests through the UI, not to mention its potential fragility (the UI is frequently adjusted throughout development). For those who wish to test just behind the UI, tools such as FitNesse[1] are very effective, whereas to test through all of the layers, tools like Selenium[2] can be utilized. Both FitNesse and Selenium are open source and freely available.

Integration Testing

Often also called *system testing*, integration testing is all about ensuring that new functionality is able to play nicely with the broader ecosystem and not just in isolation. For example, the product under development may need to integrate with other internal products such as administrative tools or with third-party products such as payment gateways.

Performance Testing

Other aliases also exist for performance testing: *load testing* and *stress testing*. The focus of performance testing is to measure the operation of the product when under pressure, typically brought about by increasing the number of users and/or increasing the amount of data processing required.

Note that unless you are actually releasing to production every sprint (or more frequently), you may not need to run performance testing every sprint. "Special-purpose types of testing such as integration testing, performance testing, usability testing, and so on may not be performed every sprint" (Cohn 2009). That being said, don't make the intervals between integration and performance test runs too long because issues found late may require a trip back to the drawing board.

1. For more information about FitNesse, go to http://en.wikipedia.org/wiki/Fitnesse.
2. For more information about Selenium, go to http://en.wikipedia.org/wiki/Selenium_(software).

Deployment Automation

If you think that getting Scrum up and running in a nice, pristine development environment is good enough, then it's time for a reality check. If your team is unable to push their work with extreme confidence, at the touch of a button, out of the nice, safe development cradle into the big, bad production environment, then you should consider your delineated phased approach to be another form of Scrummer-fall. Simply put, your team must do its utmost to completely automate the build-deploy process for all environments being used.

With that in mind, let's look at the typical key environments and how they relate to the different automated tests.

Development Environment

Earlier in the shortcut, we focused on the benefits of the CI server and the fact that it should be able to automatically run a build after every check-in.

In addition to running the CI build, I recommend running a secondary build in the development environment (traditionally known as the *nightly build*) that is triggered manually and less frequently. The difference between the CI build and the secondary build is that the latter should be given the luxury of time, and therefore, it can include the full set of tests (including all of the heavier functional and UI tests that take significantly longer to run). Shore and Warden (2007) point out some further functions that the secondary build should perform:

> In addition to compiling your source code and running tests, it should configure registry settings, initialize database schemas, set up web servers, launch processes—everything you need to build and test your software from scratch without human intervention.

The big trick here is to ensure that the development environment resembles the staging and production environments as closely as possible. For various reasons, such as licensing and speed, you may not be able to replicate all associated components, but you should still be able to stub them out using mock objects to at least simulate reality. Also, even if it is unlikely that you would replicate the entire production dataset (say, to save time, if it is of considerable size), the dataset used in the development environment should be at least representative of the real one as far as data integrity is concerned.

Why is this effort so important? By performing repeated dress rehearsals of your release into production, your system testing is happening every day rather than at the end of the release when remedial time is considerably limited.

Staging Environment

Although the development environment should at least mimic the production environment, your staging environment should be *identical* to it. The full dataset as well as all applicable third-party products and components that the product interacts with should be represented here.

In this environment, further integration testing can take place, and it is also often the primary location where performance tests are run.

Continuous Delivery and Scrum

A growing number of people in the agile community are adopting approaches such as *continuous deployment* or *continuous delivery*. Although the terms are often used interchangeably, Jez Humble (2010), coauthor of *Continuous Delivery*, explains the difference between the two:

> While continuous deployment implies continuous delivery the converse is not true. Continuous delivery is about putting the release schedule in the hands of the business, not in the hands of IT. Implementing continuous delivery means making sure your software is always production ready throughout its entire lifecycle—that any build could *potentially* be released to users at the touch of a button using a fully automated process in a matter of seconds or minutes (Humble 2010).

So, while continuous delivery makes every build *releasable* to users, continuous deployment actually *releases* every change to users (sometimes multiple times a day).

I mention these approaches because I wish to dispel a couple of myths. The first myth is that Scrum and continuous deployment/delivery are mutually exclusive. In some quarters there appears to be the perception that if you use Scrum, you can release only at the end of the sprint. This is not the case. Scrum talks about having releasable product increments at the end of the sprint, but that doesn't mean you can't also release *during* the sprint—simply make it part of the definition of done (if it applies across the board) or part of the acceptance criteria for a specific user story if it has release urgency.

The other myth I hear frequently is that some people believe Scrum dictates that you *must* release to production at the end of every sprint. Again, this is not true, and those who believe it need to appreciate the difference between the words *release* and *releasable*. Scrum does not say that you must release at the end of every sprint, but it does say that you should do everything possible to have something releasable at the end of a sprint.

Every Journey Begins with But a Small Step

Just start somewhere. I know that even after reading a brief shortcut like this one, you might feel that it all sounds too hard. If that is how you're feeling, please realize that in the case of automation, something really is better than nothing. One unit test is better than no unit tests. An automated build that takes 30 minutes is better than one that takes hours of manual work.

I recommend that if you are new to automation, you allocate a percentage of your sprint capacity to chipping away at it. Start with CI. Then automate the rest of your builds. Next, focus on applying unit tests to all new critical code, and then expand to all new code. Your next step could then be retrofitting old code with some more all-encompassing functional tests.

The choice is yours; however, either way, start somewhere and remember that without automation, you lose time, and worse than that, you are relying on fallible and time-constrained humans. To reinforce Shore and Warden's earlier quote regarding perfection, I close this shortcut with a classic from Frederick Brooks, author of *The Mythical Man-Month* (1995):

> Human beings are not accustomed to being perfect, and few areas of human activity demand it. Adjusting to the requirement for perfection is, I think, the most difficult part of learning to program.

Wrap Up

The three shortcuts included in this chapter focused on a selection of tactics, tools, and tips to help your team track and manage defects on a Scrum project. Let's recap what was covered:

Shortcut 16: Bah! Scrum Bug!

- Differentiating between bugs and issues
- A selection of underlying defect-handling principles to consider
- Approaches for tracking and managing defects within sprints

Shortcut 17: We Still Love the Testers!

- The evolution of the tester role
- Why specialized testers are still vital for high-performing Scrum teams
- Key functions that a tester should focus on: consulting, designing. and exploring

Shortcut 18: Automation Nation

- The importance of automation to avoid slipping back into waterfall patterns
- A selection of automation starting points
- How Scrum and continuous delivery play nicely together