Talking About Quality

Kathy Iberle
Iberle Consulting Group, Inc.

kiberle@kiberle.com

Abstract

Have you been tripped up by unspoken requirements? Product owners who can't articulate or simply don't talk about their expectations for usability, reliability, understandability, and all those other "ilities"? Do you know some of your "nonfunctional" requirements, but worry that you've missed some? Or perhaps your project has gotten those expectations on the table, but you're unsure whether to handle them as a user story, or in your Definition of Done, or in some other way.

This paper demonstrates how to solve these quandaries by combining traditional tools for defining quality targets with modern agile methods. We'll present a checklist of nonfunctional requirements, a set of agile tools for incorporating those attributes into your project, and some criteria for deciding which tool to use in which circumstances. As a bonus, we'll also explore how to surface hidden quality requirements by developing a defect severity chart.

Biography

Kathy Iberle is the Principal Consultant at Iberle Consulting Group, Inc., where she helps clients improve their productivity and quality. Her extensive background in process improvement and quality methods enable her to blend classic quality control with the best of current Lean thinking.

Kathy Iberle has been working with software quality and development process improvement in both agile and traditional teams for many years. After a long career at Hewlett-Packard as a programmer, quality engineer, and process improvement expert, she is now the principal consultant and owner of the Iberle Consulting Group. She has published regularly since 1997, served as co-chair of the Program Committee of the Pacific Northwest Software Quality Conference (PNSQC) in 2009, and participated in the invitation-only Software Test Managers Roundtable for five years.

Kathy has an M.S. in Computer Science from the University of Washington, and an excessive collection of degrees in Chemistry from the University of Washington and the University of Michigan. Visit her website at www.kiberle.com

1 Introduction

Customers and developers spend most of their time talking about features and functionality – in other words, what the system will do. After all, that's why people buy your system, right? If your agile team uses good testing practices and a solid Definition of Done, your system should also have all those other characteristics such as availability, reliability, installability, usability, and so forth, as if by magic.

Ah, but often the magic doesn't quite happen. After a few weeks or months of building individually delightful user stories which pass all their acceptance tests, the team discovers that the performance has degraded. Or the users can't find their way through the numerous screens. Or the system starts hanging sporadically. What's gone wrong?

Usually the problem traces back to a lack of awareness regarding the nonfunctional requirements for this product. *Nonfunctional requirements* define not **what** a system does, but how **well** it does it. Nonfunctional requirements describe expectations for system-level characteristics or *quality attributes* such as availability, usability, reliability, security, performance, and so forth. A nonfunctional requirement states how much of a particular attribute is desired. How fast should the system respond? How comfortable should a particular type of user feel?

Products fall short of meeting expectations regarding quality attributes for two main reasons:

1) Nobody ever nailed down the nonfunctional requirements – the goals for the quality attributes. The team didn't have a clear definition of what the stakeholders and users expected, so they didn't drive the design and the code to meet the users' expectations.

2) The team understood the expectations, but didn't successfully capture them in any structure or process, so the expectations were overlooked or forgotten.

The first problem involves talking about *quality* – clearly stating the expected behavior. The second problem involves talking about your *quality process* – how your team will make the expected behavior happen. In this paper, we'll talk about both.

2 Talking about Quality

If you simply ask your product owner for their nonfunctional requirements, you'll probably get a blank stare. And if you ask about quality expectations, most likely you'll hear a discussion of either defect counts or test coverage. None of these are an effective way to discover what the users actually want.

We need to understand what quality means to the users for this product. How would the users recognize quality? What level of performance, or scalability, or reliability, or usability should the product have? And how can we tell whether the product has that?

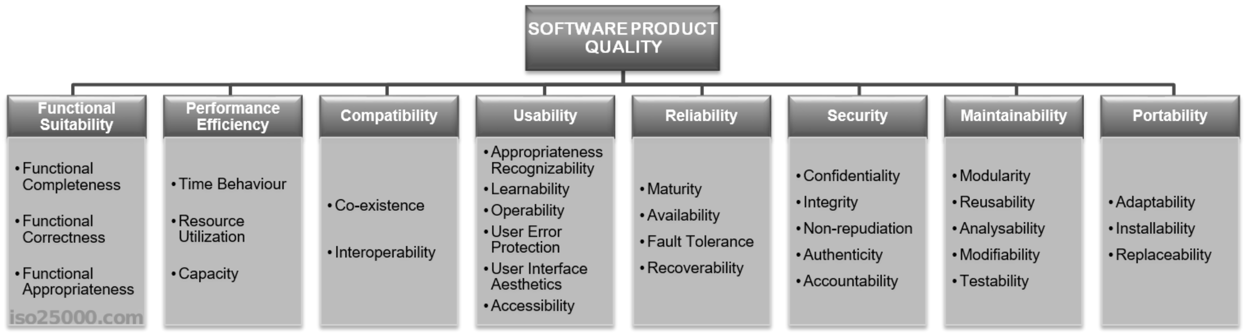
2.1 Establish a Common Language for Quality Attributes

Having a conversation with your stakeholders about their expectations for quality attributes is much easier if you have some common language. The team can provide the stakeholders with some language by using a quality attributes list.

The following list is adapted from chapter 14 of Karl Wiegers' excellent book *Software Requirements, Third Edition* [WIEG2013].

Accessibility	How easily people with a range of physical abilities can use the system.
Availability	The extent to which the system's services are available when and where they are needed.
Installability	How easy it is to correctly install, uninstall, and re-install the application.
Manufacturability	How easy it is to correctly produce copies of the software on intended media.
Integrity	How well the system prevents information loss and preserves data entered into the system
Correctness	For systems which analyze data or make predictions, how accurate is the answer provided
Performance	How quickly and predictably the system responds to user inputs or other events
Reliability	How long the system will run before experiencing a failure
Robustness	How well the system responds to unexpected operating conditions
Safety	How well the system protects against injury or damage
Security	How well the system protects against unauthorized access to the application or data
Usability	How easy it is for people to learn, remember, and use the system.
Interoperability	How easily the system can interconnect and exchange data with other systems or components.
Efficiency	How efficiently the system uses computer resources.
Scalability	How easily the system can grow to handle more users, transactions, servers, or other extensions

You can find similar lists many places, organized in various ways. You might see lists called *nonfunctional requirement types*, *quality characteristics*, *quality attributes*, *capabilities*, or even *the -ilities*. The most complex collection is the ISO/IEC standard 205010 [ISO2010], which organizes quality attributes into eight quality characteristics and thirty-one subcharacteristics.



Since the early 1980s, system test strategists have used a similar list of *system test types* to structure test coverage and brainstorm system tests.

All these different lists are heuristics or shortcuts for finding overlooked nonfunctional requirements. Most are not intended to be exhaustive or complete. They are tools to trigger thinking about important requirements which might have been missed. The definitions are part of the trigger, to help ask "how well does our product do this?"

2.2 Skip Attributes Which Aren’t Relevant

The quality attributes list is meant to trigger conversations. For your product or business, there are probably some quality attributes you don’t need to discuss. Either they aren’t relevant, or they are so central to your product that they’ve been extensively discussed already. On the other hand, if your organization usually overlooks particular attributes (and is sorry later), add those attributes to your list.

Sue’s team is developing the electronic brake control module (EBCM) for a major auto manufacturer. As part of their initial project startup, they sit down to review a list of quality attributes looking for things they’ve missed.

> Sue: “How about scalability?”
>
> Bitra: “The ability to handle more users, transactions, or whatever? That’s not relevant – our firmware has to do its job, but the job isn’t going to grow.”

The rest of the team agrees, so they drop “scalability” from their quality attributes list. They won’t ask their stakeholders for expectations in that area. But if Sue’s team was working on an Internet service, they would definitely ask about scalability.

Before removing an attribute from your list, consider all your stakeholders, not just the end users. Some of your other stakeholders might include buyers, operations staff, or maintenance staff. For instance, the operations staff of a utility company is usually quite interested in the installability of customer-facing applications, because they have to install them onto the company’s servers. The utility’s customers aren’t even aware of the installation, so they aren’t going to have useful opinions on how much installability is needed.

A list of quality attributes forces the team to look at the system from different perspectives. For instance, one might think that the attribute “availability” is only relevant to Internet services. But if one component in an app is unavailable to another component because the first component is off doing garbage collection, that can cause a huge quality problem. Asking about “availability” might surface this simply because you’re looking at the system from a different angle.

2.3 Ask the Customers What Matters to Them

A quality attributes list helps you have productive conversations with your stakeholders.

Sit down with a stakeholder and walk through your quality attributes list, asking them which attributes matter to them. When the stakeholder identifies an attribute, ask for specific goals in that area. You might find these questions useful:

- “What do you want?”
- “How will you know when you have it?” These lead to acceptance tests.
- “What will that do for you?” Understanding the relative value of different attributes helps when there are trade-offs to make.

2.4 Get Specific

In agile projects, goals for quality attributes are usually captured as acceptance criteria, tests, or release criteria. In waterfall projects, the goals are captured as written requirements. In either case, vague statements such as “the system must be fast” or “the system must be user-friendly” aren’t very helpful. The development team needs measurable, testable specifics, so they can tell when a story is “done” or when the software is “good enough” to release.

Johanna Rothman shares some examples of “good enough to release” in her book *Create Your Successful Agile Project* [ROTH2017]. These “Release Criteria” are high-level goals which were set by stakeholders.

- *“Performance*: For a given scenario (Describe it in some way), the query returns results in a maximum of two seconds.
- *Reliability:* The system maintains uptime under these conditions (describe the conditions).
- *Scalability*: The system is able to build up to 20,000 simultaneous connections and scale down to under 1,000 connections.”

Notice that these are measurable, testable statements of expectations. The scenarios and conditions are critical parts of the requirements. There are plenty of examples of such specifics in any standard requirements text. It’s worth being familiar with some of these even if you never write a formal requirements document, because they help you ask interesting questions. For instance, Karl Wiegers points out that it can be helpful to explore the attribute from multiple perspectives [WIEG2013]:

- What would users consider an unacceptable response time for this query?
- What times of the day, week, or month have much heavier usage than usual?
- How many simultaneous users do you have on average?

Expected performance attributes for the EBCM look somewhat different than performance attributes for an Internet application, but they follow the same principles. The EBCM firmware responds to “events”, which are triggered either by end user actions such as applying the brakes or by measurements delivered on a regular basis from other system components, such as sensors measuring speed. Sue’s team might put together a chart of performance requirements such as the one below.

Event	**Must respond to event within…**	**Must not respond in less than…**	**Notes**
BRAKE01	2 msec	0 msec	
BRAKE02	3 msec	0 msec	
SRM01	2 msec	0 msec	
SRM02	4 msec	0.5 msec	Too fast a response can cause mechanical stutter

2.5 Landing Zones

In some cases, there is a degree of flexibility in the requirements. Often this is a tradeoff situation, such as smartphone weight versus battery life. If the battery life is significantly better, it may be ok for the phone to be a bit heavier. In these cases, the stakeholders may want to delineate a “landing zone”, such as in this example from Rebecca Wirfs-Brock [WIRF2011].

Landing Zone for Smart Phone

ATTRIBUTE	MINIMUM	TARGET	OUTSTANDING
Battery life – standby	300 hours	320 hours	420 hours
Battery life – in use	270 minutes	300 minutes	380 minutes
Footprint in inches	2.5 x 4.8 x .57	2.4 x 4.6 x .4	2.31 x 4.5 x .37
Screen size (pixels)	600 x 400	600 x 400	960 x 640
Digital camera resolution	8 MP	8 MP	9 MP
Weight	5 oz.	4.8 oz.	4.4 oz.

The team is expected to aim for the target values, but has the option to accept a minimum value in one attribute in order to achieve outstanding in another.

This method was pioneered by Tom Gilb in the 1980s [GILB1988] using the terms “Goal, Stretch, Wish”, and has since appeared in other places using terms such as “Must, High Want, Want”, or “Must, Want, Wish”. These older sets of terms are sometimes misinterpreted, resulting in the team aiming for the lowest value rather than the middle value. In contrast, “Minimum, Target, Outstanding” clearly says: “You should aim for the target, and accept minimum only if you have to. And a gold star if you reach outstanding!” The terms themselves contribute to a productive conversation with stakeholders.

In addition to the usual pass/fail results, the team will probably be expected to report the observed value for each landing-zone attribute. This tells the organization exactly where the system has landed within the defined landing zone. If the economic return on each attribute is known, trade-off decisions can be made. See Wiegers for a systematic method to make such quality attribute trade-offs. [WIEG2013]

2.6 Ask the Business What Else Matters

Quality attributes which matter mainly to the development and maintenance teams are sometimes called *Internal Quality Attributes*. A shortfall in internal quality attributes is often considered to be technical debt rather than a defect per se.

Some Typical Internal Quality Attributes	
Modifiability	How easy it is to maintain, change, enhance, and restructure the system
Portability	How easily the system can be made to work in other operating environments
Reusability	To what extent components can be used in other systems
Verifiability or Testability	How readily developers and testers can confirm that the software was implemented correctly.

Expectations around internal quality attributes can affect how you choose to do your development, so it’s important to understand those expectations early in the project. Let’s see what these four internal quality attributes mean to Sue and her team.

After a lively discussion, Sue's team writes down list of attributes, associated requirements, and their action items to meet those requirements.

Quality Attribute	**Our Requirements for this Attribute**	**What we will do**
Modifiability Requirements	We want to refactor easily and safely, so we want: • Clear, complete documentation of API contracts. • Compliance with our company's "Future Maintenance Standard" (which covers revision control, storage of tests, coding style standards). • Enough low-level tests to quickly detect whether we've broken a module • Enough documented, repeatable tests to detect whether system is staying stable.	Include in our Definition of Done: • API documentation review. • Review of compliance with our "Future Maintenance Standards" • Has documented unit tests (coverage TBD) Create some automated end-to-end regression tests
Portability Requirements	We don't see any beyond the "Future Maintenance Standard"	n/a
Reusability Requirements	None that we know of.	n/a
Verifiability or Testability Requirements	Testability requirements from other subsystems. Verifiability regarding correct ABS responses.	Go ask the other teams. Already covered in stories.

A project may also have other miscellaneous nonfunctional requirements or constraints. Perhaps the head of the Information Technology department has forbidden use of Gnu GPL (General Public License) open source code. Or required that any use of open-source code must include capturing the exact origin and license. These types of requirements must also be taken into account in a similar fashion.

3 Quality Process: Capture and Implement the Expectations

So now you have a bunch of nonfunctional requirements. What do you do with them? In a Waterfall project, you'll capture them in a Software Requirements Specification (SRS), and then start writing acceptance tests to verify that the system meets these requirements.

In an agile project, you won't write an SRS. But user stories aren't necessarily the answer either.

In agile, we have more than one way to capture expectations and make sure they happen. User stories are the primary method, particularly for functionality, but we also capture expectations through story acceptance criteria, tests, Definition of Done, and Release Criteria.

Let's look at how to decide which method to use for a particular requirement.

3.1 Why Not User Stories?

User stories do two things for us:

1) capture the user's point of view, ensuring we understand the purpose and value of a feature

2) split the work into small, independent sections so we can get immediate feedback from users on each section

Many nonfunctional requirements can be written in user story format, such as “As an admin, I want to receive a response to any input in less than a second, so I can get on with my day.”   That captures the user’s point of view nicely. But it doesn’t split the work into small, independent sections. This story will affect many, many other user stories, and the story isn’t implementable on its own. A story which affects many other stories is known as a *crosscutting story*.

Behavior related to a quality attribute very often results in a crosscutting story. Crosscutting stories don’t help you manage your work, but instead create confusion.

3.2 Capture Expectations as Acceptance Criteria or Definition of Done

Since crosscutting user stories aren’t a good idea, what else can we do? One answer is a *quality gate*. A quality gate checks whether a story, or group of stories, or the entire product meet expectations.

All of these are quality gates:

- Acceptance Criteria for specific stories. These are often captured in the form of an acceptance test. For instance, “One-month sales data can be displayed within 1 second.”
- Definition of Done, which applies to a group of stories. This is often captured as a regression test run during every sprint. For instance, “Every story related to data display must pass Test B which demonstrates that screen response time is still 1 second.”
- Team Operating Agreement. “Everyone shall abide by our security coding standard.”
- Release Criteria. Release criteria are acceptance criteria for the entire product. While many release criteria are quality-related measurements such as “zero open high-priority defects”, some projects capture system-wide behavior specifications as release criteria, as in the example in section 2.4.

Quality gates support the experimental, iterative approach which is central to agile. Literally, we “Do the simplest possible thing that could possibly work.” [JEFF1998] and then test to see whether it passes the quality gate.

Back to Sue’s team. They’re looking at some more quality attributes. Let’s see how they apply these ideas.

> Bitra: “’Integrity: How well the system prevents information loss’. Isn’t integrity mostly a capability of databases? Maybe it doesn’t apply to our module.”
>
> Sue: “Does the EBCM firmware capture and store any data at all?”
>
> Bitra: “Well, there’s some logging. The logs are used when something goes wrong. We’ve got user stories about the logging already. The diagnostic codes go into nonvolatile memory, of course.”
>
> Tom: “Remember that space probe which failed because its logs overflowed? Maybe we should have some acceptance criteria for the logging stories which test the failure modes. We could just put a note in the user story now and add the criteria when we work on the story.”

In this case, the expectations for the attribute “Integrity” led to acceptance criteria for just a handful of stories. Sometimes that happens. Other times, a great many stories are affected.

For example, performance is almost always crosscutting. Sue’s team has defined performance requirements for each individual user story in the acceptance criteria, but will that be enough?

Bitra: "I know each user story already has acceptance criteria to ensure adequate response time, but what happens if implementing a later story makes an earlier story's response time longer? Do we have enough regression testing to notice this?"

Sam: "What if we pulled together all the tests for response times and put them in one big test? We could put that in our Definition of Done so we run it every sprint."

Tom: "Wow, that would be a big test. We'd have to automate it for sure. But then the test itself would define the acceptable performance, right? That'd be good. We could review the test with the communication harness team, to make sure they agree with us on the response times."

When it would make more sense to run the test once per sprint instead of once per user story, the test is often put into the Definition of Done rather than into acceptance criteria for individual stories. This can be appropriate for regression tests which haven't been finding many problems, or tests which require an expensive or time-consuming setup. Sue's team has used both acceptance criteria and a Definition of Done as quality gates.

For more discussion of a Definition of Done, see: http://www.jamesshore.com/Agile-Book/done_done.html

3.3 Use Standard Practices When Appropriate

Standards such as architecture, design guidelines, and coding standards take a more prescriptive approach than quality gates. Standards say: "Do the work *this* way so the end product *will* be good enough".

In a traditional waterfall project, quality gates are much more widely spaced than in an agile project. Problems accumulate between these quality gates. The organization tries to prevent the issues with layer upon layer of standards.

Agile methods depend on their frequent quality gates, but there is still a place for standards. Sometimes it is obvious that you will have to do certain things in certain ways. For instance, your security requires protecting against SQL injection. That in turn requires following certain coding practices. If you know this up front, it would be rather wasteful to **not** follow those practices in multiples stories and then go back and refactor. Necessary standards are usually captured in either the Definition of Done or the Team Operating Agreement.

Whether you are doing agile or waterfall, your decisions on whether a standard is necessary should be based on data, not "everybody knows it should be done this way". In Sue's company, the maintenance teams created their Maintainability Standard to prevent specific problems they'd encountered in the past when code was "thrown over the wall". If no one can remember why a particular standard is necessary, it may be time to question it. An experiment or spike can sort out which design guidelines, coding standards, or architecture decisions are actually necessary.

For more details on the use of standards versus quality gates, see Neal Herman's 2016 paper "Implementing Agile in an FDA Regulated Environment" [NEAL2016]. It's an interesting tale of their transition from a standards-based process to an agile process.

3.4 Not Sure? Use a Spike

Sometimes it's not clear what "good enough" would look like. Or it's not obvious what type of test would detect "not good enough", so you can't write the acceptance criteria. Or you suspect that a particular design choice isn't going to meet your stakeholder's expectations, but you're not sure.

In these cases, a spike or experiment is very helpful. A spike is a time-boxed bit of programming which is intended to answer a particular question. Once the question is answered, the spike stops. Sometimes the code is thrown away, sometimes it is integrated into the product, but the end goal of the spike isn't code. The end goal is knowledge – the answer to the question.

James Shore presents an example of a technical spike here: http://www.jamesshore.com/Agile-Book/spike_solutions.html. [SHOR2010].

3.5 If It Is a Feature, Create a User Story

Sometimes a quality attribute review turns up something which **can** be captured as an independently implementable user story.

Sue's team continues through their quality attributes list, looking for more things they might have missed. They find two features.

> Sue: "Let's take a look at what else we might have forgotten. Let's see, how about installability?"
>
> Bitra: "The car comes with the firmware already loaded into the module. So installability isn't relevant."
>
> Tom: "What about the original installation of the firmware into the module?"
>
> Bitra: "That sounds like manufacturability, not installability. I think the manufacturing engineers will want the firmware installation to be automated."
>
> Sue: "It doesn't matter what we call it, as long as we capture it. Do we have a new epic here?"
>
> Tom writes: *"As a manufacturing engineer, I can automate installation of the firmware onto the chip*."
>
> Joe: "Wait – don't we issue updates sometimes? Which would be installed by service techs?"
>
> Tom: "The EBCM isn't supposed to be serviceable, it just gets replaced."
>
> Sue: "Hmm, if there was a warranty issue, would the company ever want to do a firmware update rather than a replacement? Let's put it in the backlog and ask our product owner."
>
> Tom writes a new epic: "*As a service tech, I can install a firmware update, so the car is fixed under warranty.*"

Notice that the two user stories just written by this team are quite ordinary. There's a specific action, and a specific result. The story doesn't cross-cut other stories. The question "How much installability do you want?" doesn't even make sense. That's how we know they are valid user stories.

A quality attribute list is a good place to keep track of aspects of the system which are often overlooked in a particular organization, whether or not those are literally "system-level characteristics". Installability, manufacturability, configurability, and interoperability are often found on quality attribute lists yet can usually be captured in garden-variety user stories.

4 Nobody Wants to Play? Try a Defect Severity Chart

Sometimes, you're at the other end of the spectrum. Not only is no one interested in a systematic and thorough approach, but you can't even get the stakeholders to sit down and look at a quality attribute list, or discuss how good is good enough.

This lack of engagement can be addressed by starting at the opposite end of development, with test failures. Assume you will have defects, and ask which types of defects are serious and which aren't. Sometimes this approach is easier for stakeholders who have little development experience.

Their answers can be captured in a "Defect Severity Chart", which is an objective way to decide on the severity of a defect.

A defect severity chart is specific to the product and project for which it was written, because it defines what is considered a “serious” problem for this type of product and this group of users. That means the chart is a form of requirement. For instance, while potential injury to a user will probably be treated as a critical defect in all fields, the tolerance for intermittent failures or screen layout problems varies quite a bit depending on what type of product you are making and for whom.

Creating a defect severity chart of your own may help your team understand “how bad is bad?”. For example, this severity chart has a row for intermittent failures. That’s a reliability specification, albeit a rather fuzzy one. Stakeholders who have no experience with formally stated reliability targets will usually understand and be able to respond to a target stated in this fashion.

Sample Rubric for Grading Defects by Severity

Type of Problem	**Critical**	**High**	**Medium**	**Low**
Safety	Can injure user	--	--	--
Functionality used by 80% of users	Fail with no workaround, OR causes damage to data or system OR report incorrect data.	Fail with workaround, requires support call	Fails with obvious workaround	--
Functionality used by 20% or fewer of users	Causes damage to data or system OR report incorrect data.	Fail with no workaround	Fail with workaround, requires support call	Fails with obvious workaround
Intermittent Failure	Causes damage, or fails more often than 1/day	Happens 1/day	Happens 1/week	Happens 1/month
Internation - alization	Screen is unreadable. Language considered insulting in a major market or preventing importation	Screen is readable but looks very sloppy Translated text is wrong, correct meaning is not obvious to user	Text is cut off in minor ways Translated text is wrong but meaning is obvious	Minor messiness

The severity chart sometimes surfaces interesting implied requirements. For instance, if all safety problems are considered critical, has anybody thought about the various ways in which the software could injure the user? That might yield some new requirements which are intended to prevent failures. Severity charts can also reveal negative requirements. In this example, the internationalization row suggests that there is probably a list of phrases which should not appear in the user interface.

5 Conclusion

Quality problems often trace back to overlooking or shortchanging some of the system's nonfunctional requirements. Quality can be improved, and development made easier by using some simple tools and methods aimed specifically at surfacing and implementing nonfunctional requirements.

First, enable more productive conversations with stakeholders around nonfunctional requirements by using a quality attributes list. A quality attributes list provides:

1) language with which you can ask specific questions that will trigger useful feedback, and
2) a structure to prevent forgetting some area entirely

A quality attributes review at the start of a project may take only a few hours when the team is proficient with both the list and their subject area. If the risks, technologies, or customer expectations are quite new to the team, there is considerable disagreement concerning the expectations, or the project is quite large, the process may take as much as a few days.

If your organization has a system test team which is separate from the development team, it is definitely worth asking your system test team what they are doing regarding nonfunctional requirements. Frequently they are also asking stakeholders about quality attributes in order to develop a system test strategy. Combining efforts will likely save both groups some time. The overlap is more obvious on traditional waterfall projects using the V-model, because one of the early steps in the V-model is writing high-level acceptance tests based on the functional and nonfunctional requirements, which means the test team is out looking for those requirements even if they are not written down.

Second, systematically consider how you will ensure your product meets the nonfunctional requirements. Agile projects have more tools to employ than the team may realize. Different tools are more effective for different requirements, so you will probably use more than one.

- For non-crosscutting requirements, user stories and their acceptance criteria.
- For crosscutting requirements, quality gates at various levels:
  - Story acceptance criteria.
  - Definition of Done at either story or sprint level
  - Release criteria.
- Standard operating procedures captured in a Team Operating Agreement.

Waterfall projects use quality gates as well, albeit much more widely spaced, and usually make heavy use of standard operating procedures. The relationship between tests and requirements is less obvious on a waterfall project than it is in an agile project, so a requirements traceability matrix might be used to keep track of all the details.

Lastly, a defect severity matrix can help surface hidden requirements, especially when the organization is not enthused about a systematic review.

These methods are fairly simple to implement and can dramatically reduce unpleasant surprises late in a project. I hope you have seen at least one method which will help you talk about quality more effectively with your stakeholders and your teammates.

6 References

Gilb, Tom. *Principles of Software Engineering Management*. Addison-Wesley. 1988.

Herman, Neal. 2016. "Implementing Agile in an FDA Regulated Environment". Agile Dev/Better Software DevOps West 2016. https://www.stickyminds.com/presentation/implementing-agile-fda-regulated-environment-0 (accessed 7/14/2017).

ISO 2500 Standards -> ISO 25010. http://iso25000.com/index.php/en/iso-25000-standards/iso-25010 . downloaded July 1, 2018.

Jeffries, Ron. 1998. https://ronjeffries.com/xprog/articles/practices/pracsimplest// (accessed 6/12/2018).

Rothman, Johanna. 2002. "Release Criteria: Is This Software Done?" *STQE Magazine*, March/April 2002. https://www.jrothman.com/articles/2002/03/release-criteria-is-this-software-done/

Rothman, Johanna. 2017. *Create Your Successful Agile Project*. The Pragmatic Programmers.

Shore, James. "The Art of Agile Development: Spike Solutions". http://www.jamesshore.com/Agile-Book/spike_solutions.html. Dated 6/04/2010, accessed 7/24/2018.

Wiegers, Karl and Beatty, Joy. 2013. *Software Requirements, Third Edition*. Microsoft Press.

Wirfs-Brock: http://wirfs-brock.com/blog/2011/07/20/introducing-landing-zones/ (accessed 6/15/2018).

And if you'd like to know more about electronic brake control modules, here's a glimpse: https://www.autozone.com/repairguides/GM-Century-Lumina-Grand-Prix-Intrigue-1997-2000/ANTI-LOCK-BRAKE-SYSTEM/Electronic-Brake-Control-Module-EBCM/_/P-0900c152802180a3 (accessed 6/15/2018).