Building in Quality – Ten Years Later

Kathy Iberle
Iberle Consulting Group, Inc.

kiberle@kiberle.com

Abstract

We've all heard that it's better to "build in quality" than to test in quality. Have you ever wondered how exactly *is* quality built in?

Despite the quality shortfalls we often see, the steps to build in quality are not a complete mystery. Teams creating high-reliability software in fields such as aerospace, telecommunications, and medical applications have been building in quality for decades, and doing so for a very reasonable price. However, most of the methods for building in quality were originally designed for waterfall development.

In this talk, Kathy Iberle will revisit her 2007 perspective on building in quality and reflect on what's been learned through the agile movement. She'll update the methods, the approach, and the tools to fit today's agile projects.

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

Ten years ago, I presented a paper at this conference named "Building Quality In". The paper described how software organizations in high-reliability fields achieved high quality levels at a reasonable cost. When a failure can kill people, relying mostly on testing is too risky and too expensive. While large business projects struggled with enormous defect backlogs and crushing design problems, teams in aerospace, medical, and telecommunications were quietly building systems that worked, usually at less expense.

The primary tools used in those high-reliability projects were designed for waterfall projects. Today, with a much more complex environment, are these methods still relevant? Still useful? And how do they fit into today's agile development?

Let's take a look at how quality was achieved back in the waterfall days of 2007 and before. Then we'll jump to the present and reflect on what to bring forward and what to leave behind.

2 Building in Quality – the Waterfall Era

2.1 How Defects are Created

Why do we have defects in the first place? Basically, because someone makes a mistake during the complex and lengthy process of building an application.

We start by collecting the users' wants and desires. During this activity, we can introduce defects by:

- Misunderstanding the needs and wants stated by the user.
- Assuming the user is able to tell the team what they need. (The user often cannot articulate it.)
- Failing to ask the user about certain topics.
- Misplacing or losing the users' wants and desires.

Then we turn the users' desires into a description of features and attributes which will meet those desires. Whether that description is a collection of user stories or a traditional Software Requirements Specification (SRS), we can introduce more defects by:

- Omitting some of the user's desires.
- Writing incomplete or confusing specifications.
- Misplacing or losing some of the specifications.

Then we design software which will deliver the planned features and attributes. What will the structure of the system be, what libraries shall we use, how do the components interact? We can introduce yet more defects during this activity by:

- Making logical design errors – omitting cases, writing a state machine with incorrect transitions, insufficiently specifying interfaces, and so forth.
- Failing to implement some of the features.
- Failing to cover *unstated* quality attributes. (Everyone knows it shouldn't crash when I do this…)
- Ignoring possible behaviors of the environment – operating system, network, etc.
- Incorporating libraries which won't perform well enough for your needs.

And then we turn the design into code. We can create defects by:

- Misinterpreting the design.
- Making logical coding errors. (Ran off the end of the array, did you?)
- Using the wrong versions or the wrong files.

We create similar errors during integration and installation by:

- Installing the wrong files.
- Putting the wrong values in the configuration files.

2.2 The Quality Plan

Teams building high-reliability software were very aware of all these opportunities for error. It wasn't (and isn't) possible to achieve truly high quality without a solid understanding of where defects come from. The organization would thoughtfully and explicitly build a series of defenses to prevent errors or detect them very early. Those defenses were often captured in a *quality plan*.[1] The quality plan may have been a separate document, or it might have been spread across several different documents and plans, but the planning was there.

A quality plan usually focused on two main topics:

- Detecting mistakes very early via *quality gates*.
- Preventing mistakes completely with *preventative measures* and (sometimes) a *product risk analysis*.

2.3 Quality Gates

After each activity, there was a quality gate intended to find and fix mistakes made during that activity before proceeding to the next activity. The quality gates for requirements, specifications, and designs consisted of reviews and inspections of the written documents resulting from those activities.

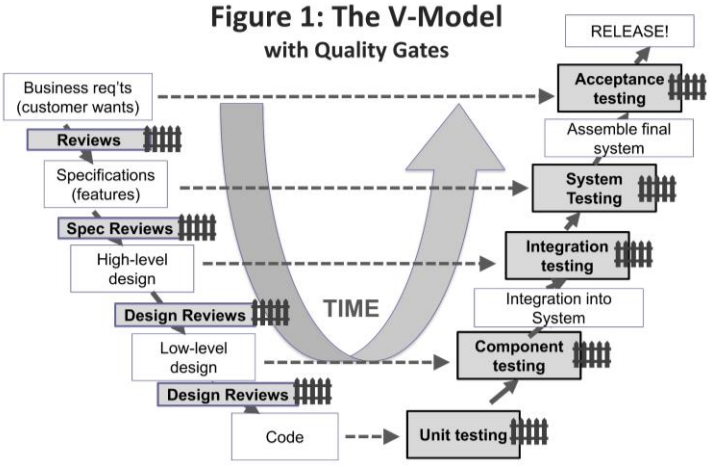
A typical quality gate for moving code into system integration on a 1990s medical instrument project:

- ❑ Compiles with no flags level 3 or above.
- ❑ Unit testing completed with 100% path coverage (ed. note: Yes, we really did that.)
- ❑ Passes all unit tests
- ❑ Integration test scripts written
- ❑ Code complexity at or below level 10
- ❑ Modules with complexity above 4 have been peer-reviewed
- ❑ Complies with coding standards

The reviews and inspections never caught anywhere near all the defects, so more quality gates were added, consisting of layer upon layer of testing.

This was popularly called the *V-model*. Mistakes made on one side of the V are found by testing on the other side of the V.

Each layer of testing was designed to find the mistakes from a particular activity, as shown by the dotted lines. For instance, unit testing is good at finding logical coding errors, but it's not effective at finding missing requirements, so you also need acceptance testing, which focuses on whether customer wants were met.



[1] To quote the American Society for Quality: "A quality plan is a document, or set of documents, which together specify quality standards, practices, resources, specifications, and the sequence of activities relevant to a particular product, service, project, or contract". (ASQ, 2017)

The final quality gate was known as *release criteria*. Since a waterfall project assumes all functionality is complete, the release criteria focused on assessing the impact of issues we already know about, and the likelihood of finding more problems later. Waterfall release criteria often looked like this list (Rothman, 2002).

- All code must compile and build for all platforms
- Zero high priority bugs open
- For all open bugs, documentation in release notes with workarounds
- All planned QA tests run, at least ninety percent pass
- Number of open defects decreasing for last three weeks

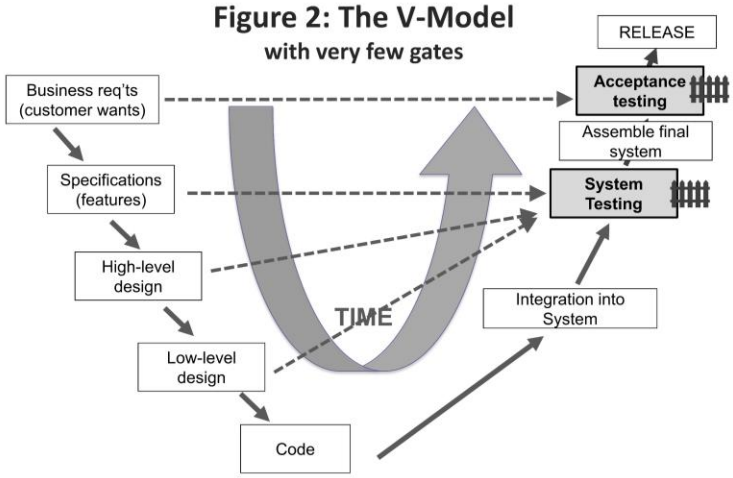
The quality gates dealt with the natural tendencies of humans to make mistakes in each activity, but they also dealt with problems caused by how the work was organized.

For instance, in some organizations making business software, each activity was performed by a specialized team, and the work then handed off to another specialized team. Different people wrote the specification, did the design, and did the coding. Each handoff was an opportunity for misunderstandings and lost information.

Some organizations responded by creating yet more quality gates requiring yet more detailed documentation. Businesses in high-reliability fields tried instead to prevent those handoff errors by creating cross-functional teams instead, where the same team was responsible for requirements, design, coding, integration, and testing. As a medical products developer, I didn't see a separate test team, let alone an entire department just for testing, until twelve years into my career.

In high-reliability fields, organizations typically used most or all of the quality gates shown in Figure 1. Outside high-reliability fields, it was pretty common to skip most of the quality gates and rely primarily on system testing, as shown to the right. Projects using this model were often known as "late".

Figure 2: The V-Model
with very few gates



Because design reviews, unit testing, and the like had not been done, system testing typically found large numbers of defects late in the project. This required a lot of rework and made the project late. Requirements gaps were a particular problem, since they were often not spotted until near the end of the project. The final quality was often problematic, because system testing simply isn't very good at finding low-level coding errors.

2.4 Preventative Measures: Building in Quality

Along with a quality gate to exit from an activity, the quality plan also defined preventative practices to be used during the activity, such as coding standards, design methods, and so forth. These were intended to prevent the developers from creating defects in the first place.

Preventative measures usually used one or more of these strategies:

- **Make the structure & logic visible**. If you can't see the structure, you **will** make mistakes. For instance, the intent of most coding standards is to make the logic easily visible.
- **Maintain intellectual control.** If you think about too many things at once, you **will** lose track and make mistakes. About seven things will fit in your head at once, so if your code or design has more than seven, decompose into smaller pieces. Work in small, verifiable steps.
- **Know your domain**. If someone has explained how to use this API, or how this service works, or how to apply this design pattern, do acquire this knowledge! A decent education in the methods and technology prevents a lot of problems.
- **Mistake-proof the routine stuff.** Use, and if necessary build, tools which won't let you make certain mistakes. FORTRAN started out in 1957 as a way to translate math formulas into code more quickly and reliably than was ever possible using assembly language. (UM 1999). You can build your own tools to prevent your own mistakes. For instance, a wizard to write those pesky configuration files.
- **Design away the opportunity for error.** Sometimes a good design choice can eliminate a certain class of errors. An example from James Shore (Shore 2010): "if you have repeated problems with UI field lengths being inconsistent with database field lengths, change your design to base one value off of the other."

I'll describe a few of the broader preventative measures which were commonly used in the waterfall era.

2.4.1 Defining and Tracking Non-functional Requirements

Non-functional requirements or *quality attributes* are those requirements which state not what a product does, but how well it does what it does. For instance (Wiegers 2013 pp 269-273)

- The system shall protect against unauthorized addition, deletion, or modification of data.
- At least 98% of the time, the trading system shall update the transaction status display within 1 second after the completion of each trade.

Typical non-functional requirements are performance, security, availability, usability, and many other "ilities". If the developer doesn't know what is expected, she can't drive the design and code to meet the target. In high-reliability fields, we always defined our targets carefully so we'd have a fighting chance of hitting them. Practices to prevent missing or misunderstanding the non-functional requirements included checklists of quality attributes to consider defining, templates, and training on how to ask customers what they need in these areas without over-promising. We sometimes wrote system tests for the non-functional requirements quite early, because writing the tests helped clarify the requirements.

2.4.2 Quality Factoring

It's great to define your non-functional requirements, and write system tests to check against those requirements, but if that's all you do, your system will fail the tests. You've got to think about how to meet those requirements.

In high-reliability fields, we traced the non-functional requirements through the design. At each level, we noted both what needed to be done in order to achieve the goal, and what should be tested at that level to ensure we were on track. Martyn Ould calls this *quality factoring* (Ould, 183). For instance, when the instrument was expected to respond in 1 second, that requirement was linked to a *performance budget* in the design for the responsible subsystem. In the performance budget, each firmware component in the response chain was assigned a portion of that 1 second, and a corresponding test was added to that component's test suite. The owner of the component might factor this further by creating a performance budget for each module in the subsystem, and so on.

Some attributes were achieved by defining system-wide criteria which linked to system-wide practices. For instance, a security attribute might require all user data entry fields to screen for commands such as "DROP TABLE". In all cases, you were expected to work through a 3-step process and factor as needed:

1) What's the goal?
2) How will the system achieve the goal?
3) How will I verify the system has achieved the goal?

In my experience, quality factoring was commonly used but rarely taught in any systematic fashion. It didn't even have a consistent name. One of the better descriptions is in Martyn Ould's book (Ould 1999). Quality factoring was similar in some ways to today's a*cceptance-test driven design* (*ATDD)*. However, since the waterfall process forced us to deal with the entire system at once, rather than a small group of stories, the sheer volume of detail to track and the long feedback loops could overwhelm the process.

2.4.3 Product Risk Analysis

Another widely used preventative measure was the product risk analysis. Early in the project, the team was asked to list the ways in which the end product could fail terribly. What disastrous damage could this product do? How could it injure a user, destroy all their data, damage your company's reputation, bring down your state's electric power grid, alienate your users?

After identifying the most likely disasters, the team looked for ways to prevent those disasters by working backwards from the disaster to its potential root causes. The team would then strengthen the quality gates with more detection measures (reviews, inspections, testing) and/or add more preventative measures to the quality plan. Some preventative measures I've seen come out of a product risk analysis:

Product Risk	**Preventative measures**
Routines which call third-party library don't work, because we didn't fully understand the third-party library.	**Know your domain**. Send team member to training class. (It's surprising how often this is neglected.)
Installation instructions in the install poster and on-screen don't match.	**Design away the opportunity for error.** Don't put the instructions in both places. Put them only on-screen. That's where most people look.
Printers don't install correctly due to minor errors in their configuration files.	**Mistake-proof your process**. Automate the writing of the configuration files with a wizard to do all the tedious predictable steps.
Device control doesn't work correctly in all cases, due to errors in a long set of nested if-then-else statements.	**Know your domain**. Design and code a state machine instead of ad-hoc if-then-else statements.

Today, the most common place to find a product risk analysis is in testing books.[2] In my opinion, this is the wrong place. Development teams should do the product risk analysis, so they can reduce risk by reducing the potential to make those mistakes. All the test group can do is find them after they're made.

2.5 What Didn't Work – Slow Feedback

Over the years, many organizations built very reliable and robust systems using these methods and a waterfall lifecycle. However, the waterfall lifecycle had one enormous drawback. Feedback was extremely slow. Mistakes were made, and nobody knew about them until months later when the problems showed up in integration or system test. That made them expensive to fix.

[2] You can find a more detailed discussion of risk analysis in many books on testing, such as *Critical Test Processes* by Rex Black [Black 2004] and resources on quality planning such as Martyn Ould's *Managing Software Quality and Business Risk*. [Ould 1999].

The mistakes with the longest feedback time were those involving understanding what the user actually wanted. That's bad because there will almost certainly be gaps in that understanding. Users aren't always able to articulate what they need, and sometimes they simply don't know. By 1999, many organizations were using multiple forms of prototyping to elicit customer needs by showing them running software or mockups of running software.

The long feedback loop was a problem in design as well, since the results of a design decision are often not visible until the system is running. Many teams prototyped parts of the design before completing the design, despite their waterfall process saying they shouldn't. In essence, the team was performing an architectural spike to investigate some worrisome aspect of the design.

I first tried an iterative method in 1993, when we used staged delivery to design, code, and deliver a medical product. Despite not using many of today's tools such as cadence and user stories, we found that simply reducing our feedback loop from eighteen months to about six weeks was an enormous help in dealing with both requirements misunderstandings and designs that didn't work as expected.

3 Building in Quality – the Agile Era

What's changed in the last ten years? Well, defects are still created by making mistakes during activities. The core activities still involve gathering user requirements, deciding what features to build, designing the system, coding it, and integrating or installing it. The biggest change in software development practices over the last ten years has been the widespread adoption of agile practices. How does this affect building in quality?

3.1 You Can't Know Everything

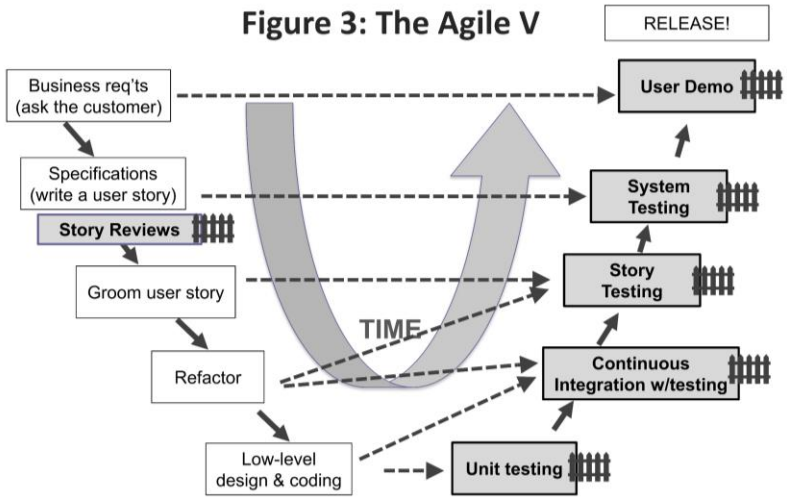
One of the fundamental concepts of agile is that you can't know everything in advance. There will be "mistakes" of omission. You won't understand everything the user wants, the environment will change under your feet and affect your design, new security challenges will arise. Rather than trying to prevent these omissions with detailed documentation up front, an agile project assumes the omissions will happen and concentrates on finding them rapidly.

3.2 The Agile V: A Single Sprint

In an agile project, the entire "V" is performed in just a couple of weeks, instead of months or years. The "V" is repeated over and over on successive user stories.

As shown to the right, there's less distinction between design and coding activities. User stories capture requirements (what the user wants and why) and the specification is captured as acceptance tests.

Figure 3: The Agile V



The quality gates meant to find omitted requirements look very different in agile than in waterfall – and they're a lot more effective. There's two quality gates for requirements – a user story review and the sprint review. During the sprint review, the user sees the finished features and can provide feedback. The time between deciding what to make and getting customer feedback on your decisions is a couple of weeks instead of many months, so the inevitable misunderstandings are small and easy to fix.

Omissions in design are also detected rapidly, if the acceptance testing is sufficiently thorough. And finally, because the same team performs all the activities in quick succession, there are far fewer handoff errors than in a waterfall process with separate teams for each activity.

3.3 Quality Gates in Agile

During initial adoption of agile, many teams continued to struggle with mistakes made during design and coding, often causing the team to miss their sprint commitments. In my experience, this is often due to insufficient quality gates and preventative practices inside the sprint itself. For instance, they'd have the sprint review but not the earlier user story review, or they'd completely skip unit testing.

Some organizations were simply carrying over their poor waterfall practices into agile. They'd relied mostly on system testing to find defects in their waterfall days, rather than focusing on preventing them or finding them early. They did the same in their agile projects, spending the first half of the sprint coding and then discovering reams of defects as soon as they started testing.

Other organizations dropped all their internal quality gates when they switched to agile, believing that the gates weren't needed inside an agile sprint. This was particularly common in groups that adopted Scrum without any exposure to XP or earlier agile methods. The Agile methods prior to the Agile Manifesto included quality gates.[3] XP insisted on several very powerful quality practices and gates: test-driven design, pair programming, and extensive unit testing. (Beck, 2000). However, Scrum initially had little to say on the topic. Many people didn't understand that Scrum was a project management structure, not a full-featured engineering practice. They assumed that if Scrum didn't teach it, you didn't need it. This resulted in dropping quality gates inside the sprint, and not putting anything in their place.

3.3.1 Definition of Done

By 2005, Scrum training sessions had started restoring those quality gates as a "Definition of Done". (Agile Alliance 2017). A Definition of Done is the exit criterion for the entire sprint. It almost always includes quality gates for individual activities within the sprint, as you can see in this Definition of Done from James Shore's *Art of Agile Development*. (Shore, 2010)

This Definition of Done is a combination of all the internal quality gates and the release criteria in a typical waterfall process. There isn't much emphasis on meeting the quality gates in a specific order, but that doesn't matter much when all the activity takes place in a week or two. The small iterations actually make it easier to convince the developers to use ***more*** quality gates. Meeting each and every criterion apparently doesn't seem so daunting when you're looking at just a single story's worth of code.

Definition of Done: (Shore, 2010)

- ❑ Tested (all unit, integration, and customer tests finished)
- ❑ Coded (all code written)
- ❑ Designed (code refactored to the team's satisfaction)
- ❑ Integrated (the story works from end to end—typically, UI to database—and fits into the rest of the software)
- ❑ Builds (the build script includes any new modules)
- ❑ Installs (the build script includes the story in the automated installer)
- ❑ Migrates (the build script updates database schema if necessary; the installer migrates data when appropriate)
- ❑ Reviewed (customers have reviewed the story and confirmed that it meets their expectations)
- ❑ Fixed (all known bugs have been fixed or scheduled as their own stories)
- ❑ Accepted (customers agree that the story is finished)

[3] For instance, staged delivery (McConnell 1996) and evolutionary delivery (Gilb 1985)

Here's a more detailed Definition of Done from a medical products company using Agile. (Herman, 2016). Despite looking a bit waterfall-ish, this Definition of Done doesn't obligate the organization to perform design and coding sequentially. They just have to demonstrate that they did everything on the list by the time the sprint ends.

The author observes that the list deliberately calls out design and coding separately so it's clear to both themselves and regulatory agencies how they are preventing or finding problems in each activity.

It's also interesting (and admirable) that they're requiring a root cause analysis for each defect found in acceptance testing.

Definition of Done: Checklist for Technical Completeness for User Stories (Herman 2016)

Design:

- ❑ Design covers everything in the user story and acceptance criteria.
- ❑ Design reviewed by area experts and feedback is incorporated.
- ❑ User story has a link to the design.

Code:

- ❑ Code implements the design.
- ❑ Unit tests cover the design (includes use cases, API contracts).
- ❑ Code compiles and runs, on the build server, without errors, warnings, or unit test failures.
- ❑ Code and unit tests have been peer reviewed and adjustments made per comments.
- ❑ No new defects.

Acceptance Testing:

- ❑ Acceptance tests in a form listed below have been written and entered into project management system.
  - ❑ May include manual, automated, and unit or integration tests
  - ❑ Verification Procedure and/or SMART for high risk stories
  - ❑ SMART for stories with medium risk
  - ❑ Acceptance tests have been reviewed by a developer and feedback has been incorporated.
  - ❑ Acceptance tests pass on a branch or main build; unresolved issues found on main build have been logged into defect tracking system.
  - ❑ Executed results have been attached to project management system.

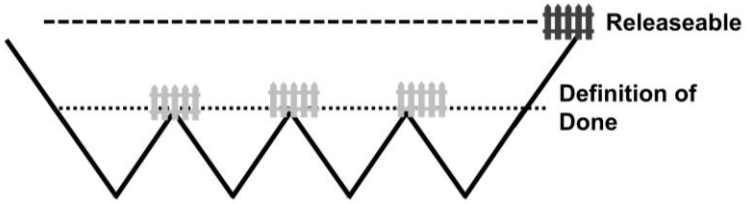
Defect Fixing:

- ❑ Minimal steps to reproduce are documented in the defect description.
- ❑ Root cause analysis is documented in the defect description.
- ❑ Fix approach is documented in the defect description.

3.3.2 More Than One Definition of Done

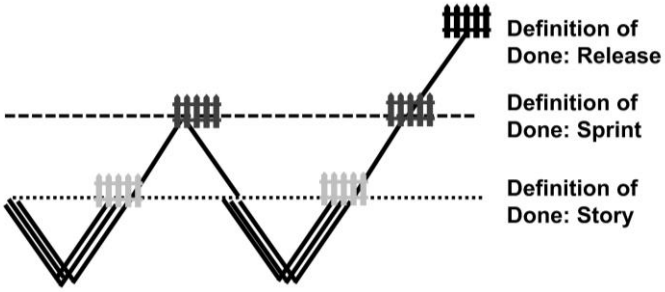
A single Definition of Done makes sense if there are no handoffs between teams inside the sprint, and each sprint results in releasable code. The process is a series of tiny V's with a quality gate at the end of each "V" or sprint.

But what if your system doesn't reach releasable quality at the end of each sprint? Perhaps your organization has a hardening sprint, system testing, or localization after the sprint is over. That process might look more like this. The Definition of Done clearly is not the same as Releasable.



In this case, you'll need additional criteria prior to release. You might call it "done-done", or you might call it release criteria, or something else again. Whatever you call it, it's a second quality gate.

It's not unusual to see organizations define three levels of "Definition of Done", as shown to the right. In this case, multiple stories are individually completed to meet the Definition of Done for a story, and then the whole group of stories is integrated together and reviewed by the user to meet the Definition of Done for the Sprint. And then further work is done on the results of several sprints to meet a Definition of Done for release. That's three quality gates.



Joseph Ruskiewicz, in his paper "Establishing a Definition of Done in Complex Organizations" (Ruskiewicz 2015), reflected on what happens when a large organization with multiple interrelated teams and a semi-agile process passes work from one team to another. They concluded that they needed quality gates for handoffs from one team to another, and that the gates shouldn't all be identical. The gates are all named "Definition of Ready" rather than "Definition of Done", to avoid confusion over what "done" actually means. The practice seemed to be:

- All groups define a Definition of Ready to ensure a level of quality is met before passing their work to another group.
- The organization mandates a few basic criteria must appear in every Definition of Ready.
- Teams add their own specific criteria, specific to their technology and known risks.

You can certainly argue about whether this is or isn't an agile process, but whatever your point of view, the organization needs to control its quality within its current process. They need those quality gates. Different Definitions of Done for different purposes makes sense in this context.

3.3.3 Continuous Integration: A Special Quality Gate

Continuous integration (CI) is the practice of checking code into the shared repository frequently, where it is compiled and subjected to some automated testing. Continuous integration is intended to find certain types of defects, so it acts as a quality gate inside the sprint. Simply compiling all the code together will find certain types of coding conflicts, but the main goal is running automated tests to find design conflicts between different individuals' or teams' code. The code is sent through this gate as often as is practical, given the tools available and the amount of automated testing.

Back in the 1990s, system-wide integration daily or weekly was not unusual on smaller projects. On large projects, building the entire system from scratch more frequently than daily often wasn't practical with the available tools, and the lack of tools and machine speed often put severe limits on the amount of automated testing which could be done.

Today, continuous (or very frequent) integration is much more practical than it was ten or fifteen years ago. Just remember that the goal is to detect problems, not just to run builds. If your builds never fail, or the tests never report any errors, you aren't getting any benefit from this quality gate. Likewise, if the tests fail but there's not enough information to find the defects causing the failure, this quality gate isn't much use.

3.4 Preventative Measures: Building in Quality

XP and evolutionary delivery both placed a lot of emphasis on preventing developers from making mistakes, rather than finding the mistakes afterwards. This has carried over into the wider agile practice. Good practices from the past were incorporated and often transformed to fit the small increments of agile.

Some of the most popular and effective practices are

- design patterns
- test-driven development (TDD) and refactoring
- acceptance-test driven development (ATDD) and its cousin Behavior-driven development (BDD)
- pairing

3.4.1 Design Patterns

Design patterns originated in object-oriented programming as a way of capturing reusable concepts in object-oriented software design. Prior to that, reusable concepts were typically captured as algorithms (which are still used today, of course) and design rules in structured programming. The typical design pattern provides:

- A pattern name.
- A pattern intent or purpose. What problem is this pattern intended to solve?
- The pattern itself – an outline of the objects or classes and their basic behaviors.
- The pattern's consequences and side-effects which may affect whether it is an appropriate solution for a given problem.

Design patterns help a developer maintain **intellectual control** by providing an abstraction using the concepts of classes and inheritance. A pattern is a simpler way to look at a complicated bunch of code. And knowing a lot of design patterns relevant to your domain is **knowing your domain**. If somebody else already solved this, why solve it again? Books of algorithms serve much the same purpose as design patterns on a more detailed level.

3.4.2 Test-Driven Development and Refactoring

Test-driven development, or TDD, is very effective at preventing errors during coding and low-level design. In TDD, developers write the unit tests first, verify that the tests fail, and then write the code and

verify that the tests pass. TDD forces you to maintain **intellectual control** by working in small, verifiable steps.

TDD can be used to great positive effect in waterfall projects as well as agile projects. Some people did practice TDD before Kent Beck wrote about it (Larman 2003), but it didn't have a name and wasn't widely known. Giving the practice a name and writing about how to do it made TDD much easier to adopt and therefore much more widespread. Today, TDD is a cornerstone of quality plans in many organizations.

Refactoring is a relative of TDD. To quote Martin Fowler, refactoring is "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior." Basically, a section of code is rewritten in small steps verified by tests. Again, we see the strategies of **maintaining intellectual control** and **making the logic visible**.

3.4.3 Acceptance-Test Driven Development

TDD prevents coding errors, but it doesn't do much to prevent missing requirements or misunderstanding the demands of the environment on the design.

An effective practice for capturing requirements and design constraints is acceptance-test-driven development, or ATDD. In ATDD, the agile team captures many of the requirements and design constraints for each story as *acceptance tests* for that story.

The old V-model actually recommended something very similar - write the requirements, and then write the acceptance tests for those requirements before beginning design. However, it wasn't as effective as today's ATDD because there wasn't as much user review of the tests (to ensure requirements were understood) and the whole practice tended to bog down under the sheer weight of trying to cover every aspect of the product at once.

3.4.4 Pairing and the Three Amigos

Pairing is another effective method for preventing defects. Two programmers might pair to write code, or install instructions, or any other deliverable. A tester and a developer might pair to write user stories. In all these cases, more than one set of eyes on the job prevents mistakes.

Johanna Rothman describes the advantages of multiple viewpoints during ATDD in *Create Your Successful Agile Project* (Rothman 2017). "When teams use the Three Amigos [to develop stories], they split into triads of one developer, one tester, and one product owner or BA. The idea is that each person has his or her unique perspective on the possible story:

- The developer thinks about how to implement the story.
- The tester thinks about what can go wrong.
- The business analyst thinks about what the customer or business wants to accomplish with this story.

… If they add acceptance criteria, especially in the form of Given-When-Then, the team is much more likely to understand the story."

3.4.5 Non-functional Requirements in Agile

Remember those non-functional requirements? The ones the customer hardly ever tells you about? When organizations stopped writing big requirements documents and started writing user stories, many of them left out the non-functional requirements. Then they were surprised when the end product didn't meet their unwritten non-functional requirements. Others realized that they needed to keep the non-functional requirements, but they weren't sure where or how.

Today, I see agile organizations handling their non-functional requirements in several different ways.

- Some groups define each non-functional requirement as a story or epic, such as “As a user, I expect all screens to respond within 1 second”.
- Other groups add the non-functional requirements to the Definition of Done for all stories, often in the form of an acceptance test. E.g. “Each story must pass Test B which demonstrates that screen response time is still 1 second.” This works well in concert with ATDD.
- SAFe treats the non-functional requirements as “constraints or restrictions on the design of the system across the different backlogs.” http://www.scaledagileframework.com/non-functional-requirements/.

All of this assumes that someone actually defined the non-functional requirements, despite the users’ usual vagueness on this topic. That’s frequently a big hole. Johanna Rothman recommends getting users involved during initial stakeholder agreements by including the non-functional requirements in the project’s Release Criteria. She gives some examples in her new book *Create Your Successful Agile Project* (Rothman 2017).

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

4 Continuous Improvement

Even with all the quality gates and preventative practices, problems still do occur. The classic response was to monitor defects found in the field and use defect cause analysis to pinpoint areas where a stronger gate or a preventative practice would help. (Iberle, 2007)

This is still a very effective method for improving the development process. Unfortunately, it is often left out of sprint retrospectives, probably because the data isn’t available until well after the sprint is over.

Any group that is really serious about improving their quality should master the art of defect cause analysis. Some hints:

- Developers, not testers, should determine the cause of the defect. The testers don’t know, so don’t ask them to do it.
- Developers need some training. Many initially think that all defects are coding mistakes. As we saw earlier in this paper, that’s not the case.
- Pairing or teaming can be much more effective than working individually.

There’s a longer description and references for defect cause analysis in my 2007 paper, which is available on my website as well as in the PNSQC archives. There’s a link in the References.

Once you know where the problems are coming from, you’ll want to fix that process. James Shore says it well: “The best way to fix your process is to make mistakes impossible. For example, if you have a problem with UI field lengths being inconsistent with database field lengths, you could change your design to base one value off of the other.

The next best solution is to find mistakes automatically. For example, you could write a test to check all UI and database fields for consistency and run that test as part of every build.

The least effective approach (but better than nothing!) is to add a manual step to your process. For example, you could add "check modified database/UI fields for length consistency" to your "done done" checklist.” (Shore 2010)

5 What We Have Learned Since 2007

The key to building in quality is still prevention of defects, rather than finding and fixing them after the fact. The strategies for preventing defects haven't changed fundamentally, but we do have more and better practices to prevent defects due to both agile development and faster machines, while many of the older practices such as coding standards remain mainstays of the profession.

Better tools and faster machines have made continuous integration practical, and there's many more tools for automating unit and regression tests, which makes refactoring easier and safer. Agile and object-oriented programming have brought us some new practices, many of which are also applicable across agile and waterfall. Design patterns, pairing, and TDD can be effective on many types of projects.

Adoption of agile with its short iterations changes the quality game for the better in several ways:

- The user story format leads the developers to think more about why the customer wants the feature, capturing requirements which were often left out in the past.
- Cross-functional teams cut down on handoff errors and increase communication.
- The short iterations provide rapid feedback on whether user needs have been correctly understood and whether the design works.
- The short iterations with less documentation make rigorous quality practices (as captured in a Definition of Done) seem feasible rather than overwhelming.

However, adoption of agile can pose challenges to achieving high quality, especially if the adoption is done "by the book" without understanding why the ideas in the book work:

- Collapsing all your quality gates into a single Definition of Done can result in dropping useful quality criteria and letting defects escape. If you have additional steps between the end of the sprint and releasability, you need at least two quality gates.
- Not understanding how to manage non-functional requirements in agile can result in losing sight of important quality attributes and failing to implement them.
- An overly strict focus on delivering immediate user value in every single user story can make it very difficult to achieve some non-functional requirements. Some system-level design work is often necessary to identify the acceptance criteria which will eventually add up to the desired non-functional requirement.

Many organizations, both waterfall and agile, still don't use either product risk analysis or defect cause analysis to identify the defect prevention mechanisms which will be most effective for them.

6 Conclusion

If you only remember one thing from this paper, I hope it will be one of these strategies to prevent defects and build quality into your software from the start!

- Make the structure & logic visible.
- Maintain intellectual control.
- Know your domain.
- Mistake-proof the routine stuff.
- Design away the opportunity for error.

7 References

Agile Alliance. “Definition of Done”, https://www.agilealliance.org/glossary/definition-of-done/ (accessed 7/20/2017).

American Society for Quality (ASQ). 2017. “Learn About Quality: Quality Plans”. http://asq.org/learn-about-quality/quality-plans/ (accessed 7/20/2017).

Beck, Kent. 2000. *Extreme Programming Explained*. New York: Addison-Wesley.

Black, Rex. 2004. *Critical Testing Processes*. New York: Addison-Wesley.

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).

Iberle, Kathy. 2007. “Building in Quality”. Proceedings of the Pacific Northwest Software Quality Conference 2007. Also available at http://kiberle.com/wp-content/uploads/2016/01/2007-BuildingQualityIn.pdf

Larman, Craig. 2003. Computer, June 2003 issue. Accessed at: http://www.craiglarman.com/wiki/downloads/misc/history-of-iterative-larman-and-basili-ieee-computer.pdf

McConnell, Steve. *Rapid Development.* Microsoft Press. 1996.

Poppendieck, Mary. Poppendieck, Tom. 2003. *Lean Software Development: An Agile Toolkit*. 2003. Addison-Wesley. Pp 142-143.

Ould, Martyn. *Managing Software Quality and Business Risk*. 1999.

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. 2007. *Manage It! Your Guide to Modern, Pragmatic Project Management*. The Pragmatic Programmers.

Rothman, Johanna, 2017. *Create Your Successful Agile Project*, Ch. 5. The Pragmatic Programmers.

Ruberto, John. 2017. “100 Percent Unit Test Coverage Is Not Enough.”, Agile Connections by StickyMinds, entry posted 7/10/2017, https://www.stickyminds.com/article/100-percent-unit-test-coverage-not-enough/ (accessed 7/17/2017).

Ruskiewicz, J, 2015. “Definition of Done”. Proceedings of the Pacific Northwest Software Quality Conference 2015.

Shore, James and Warden, Shane. 2007. *The Art of Agile Development.* O’Reilly.

Thibodeaux, Adrian. Pandya, Chintan. “End-to-End Quality Approach: 14 Levels of Testing”. AgileDev/Better Software West 2016. https://www.slideshare.net/JosiahRenaudin/endtoend-quality-approach-14-levels-of-testing (accessed 7/14/2017).

University of Michigan 1999. http://groups.engin.umd.umich.edu/CIS/course.des/cis400/fortran/fortran.html (accessed 7/28/2017).