QA to AQ Part Two

Shifting from Quality Assurance to Agile Quality
"Measuring and Monitoring Quality"

Joseph W. Yoder [1] and Rebecca Wirfs-Brock[2]

[1] The Refactory, Inc.,

[2]Wirfs-Brock Associates, Inc.

joe@refactory.com, rebecca@wirfs-brock.com

Abstract. *As organizations transition to agile processes, Quality Assurance (QA) activities and roles need to evolve. Traditionally, QA activities occur late in the process, after the software is fully functioning. As a consequence, QA departments have been "quality gatekeepers" rather than actively engaged in the ongoing development and delivery of quality software. Agile teams incrementally deliver working software. Incremental delivery provides an opportunity to engage in QA activities much earlier, ensuring that both functionality and important system qualities are addressed just in time, rather than too late. Agile teams embrace a "whole team" approach. Even though special skills may be required to perform certain development and Quality Assurance tasks, everyone on the team is focused on the delivery of quality software. This paper is part two of a series of patterns about Agile QA practices and activities. The patterns in this paper are focused primarily on measuring and monitoring system qualities.*

Categories and Subject Descriptors
• **Software and its engineering~Agile software development** • **Social and professional topics~Quality assurance** • *Software and its engineering~Acceptance testing* • Software and its engineering~Software testing and debugging

General Terms
Agile, Quality Assurance, Patterns, Testing

Keywords
Agile Quality, Quality Assurance, Software Quality, System Qualities, Testing, Patterns, Agile Software Development, Scrum, Quality Related Acceptance Criteria, Agile Quality Scenario, Whole Team

Introduction

An important but difficult task for any software development team is identifying the important qualities (e.g. non-functional requirements) for a system. Quite often these system qualities, such as reliability, scalability, or performance, are overlooked or simplified until late in the development process, thus causing time delays due to extensive refactoring and rework of the software design required to correct quality flaws.

Quality Control generally refers to inspection activities that occur at the end of a process. Quality Assurance or Total Quality Control is an alternative to Quality Control which recognizes that inspection at the end is ineffective and that you can be more effective if you take a more holistic approach that builds quality into your process from the start, engaging the whole team. Given its original intent, it is ironic, or perhaps tragic, that Quality Assurance has come to be associated with late-in-process activities performed only by QA personnel. Too many so-called Quality Assurance departments don't seem to have learned what Quality Assurance means[1]. As organizations move to being more agile, it is important that this transition also includes Quality Assurance (QA).

Typically, QA groups have worked independently from the software team. However, in agile teams, QA works more closely with the whole team, including business, management, and developers on an ongoing and daily basis, helping to ensure quality in all facets of development. Because QA is more engaged throughout, Agile QA requires additional skills. For example, Agile QA needs to understand both requirements and the code and know how to write their own automated suite of test cases.

Previously in [YWA] we introduced 24 patterns to address issues about being agile at quality and wrote patterns for 6 of them and patlets for the rest. A list of all the patlets is included in the appendix. We organize our software-related Agile Quality patterns into these categories: fitting quality into your process, identifying system qualities, making qualities visible, and being agile at quality assurance. The patterns written in our previous paper were *Integrate Quality* into your Agile Process, *Agile Quality Scenarios*, *Quality Acceptance Stories*, *Fold-Out Qualities*, *Whole Team*, and *Quality Focused Sprint*.

This paper extends our original work with additional patterns for identifying system qualities and making them visible: *Find Essential Qualities*, *Agile Landing Zones*, *Measurable System Qualities*, *Agree on Quality Targets*, *Recalibrate the Landing Zone*, and *System Quality Dashboards*. It is useful for team members to be aware of important system qualities and have them readily available.

These patterns are intended for agile teams who want to integrate QA activities into their agile processes or who are concerned that system qualities are adequately addressed throughout their project. These patterns are applicable whether or not you have separate QA teams and roles or extensive agile experience.

Our patterns are written in the spirit of Edward Deming's fourteen principles for business transformation and improvement [De]. Consequently, our patterns focus on actions for improving software quality and integrating QA concerns and roles into the whole team. Our patterns are written according to Takashi Iba's Patterns 3.0 pattern languages [IBA] for guiding human action.

[1] Inspired by an email dialog with one of our reviewers, Jason Yip.

Find Essential Qualities

"The ability to simplify means to eliminate the unnecessary so that the necessary may speak." —Hans Hofmann

Quite often essential system qualities are overlooked or simplified until late in the development process. This can cause delays due to extensive refactoring and rework of the software design in order to correct quality flaws. To avoid extensive rework it is important that agile teams identify these fundamental qualities and make those qualities visible to the team in a timely manner.

How can agile teams understand essential qualities for an evolving system?

❖ ❖ ❖

Not focusing on fundamental qualities early enough can cause significant problems, delays and rework. Correcting functional flaws can be time-consuming. Remedying performance or scalability deficiencies can require significant changes and modifications to the system's architecture.

If essential system qualities were identified and addressed during earlier sprints, significant architectural verification could be performed earlier, preventing significant disruptions or delays caused by architectural flaws.

Focusing on system qualities sometimes distracts from important functional requirements. The hard part is trying to appropriately divide your attention between functionality and system quality concerns.

On one hand it would be good if you could develop and automate tests for all system qualities. It would be great to test system qualities daily. However, for example, testing something like usability could be expensive and not practical to test so frequently. Trying to find a reasonable balance between how often the tests are ran versus the costs of performing the tests can sometimes be difficult.

❖ ❖ ❖

Therefore, have team meetings at opportune times with important stakeholders to brainstorm the most important qualities that need to be considered for the system. At the start of a project it is important to identify essential qualities critical to the success of the project. This can be done via an agile quality attribute workshop where you agree on essential qualities, and make sure they are visible to team. These workshops should include key members such as the product owner, developers, architects, quality assurance, and the

customer. Agile quality attribute workshops need not be not long, drawn out affairs. Whenever there are major changes to the roadmap or new system qualities become apparent, the team can choose to hold another quality workshop.

During a quality workshop, which might last an hour or two, simple collaborative techniques can be used to identify and characterize system qualities. A *Quality Chart* can be put on a whiteboard that team members use to note specific quality concerns. People can identify a concern and write it on a sticky note that is associated with a specific system quality (such as performance or reliability). The team can vote on what they consider most important and urgent and then write *Agile Quality Scenarios* for those. Teams can use quality sheets or templates to record the quality scenarios.

After formulating a product or project roadmap, which outlines the major features and the order that they could be delivered, potential architectural risks can be identified. Based on these risks, quality-specific concerns can be identified and tied to roadmap features. A rough timeline of when specific qualities and architectural capabilities need to be delivered that enable specific features can also be sketched out and included on the backlog.

Also, during sprint or release planning is another good time to revisit and address any new system quality concerns. Team members can identify specific system qualities that need to be delivered in the release and write *Quality Stories*, and identify activities needed to achieve them.

Essential system qualities can be monitored throughout the project with *System Quality Dashboards*.

Agile Landing Zones

"Skate where the puck's going, not where it's been" —Wayne Gretzsky

On a complex project or product, you need to be aware of those system qualities that contribute to your project's success. You don't want these essential success criteria to get lost in with the myriad of other requirements.

You also need to make design tradeoffs as you implement your system. Almost always these tradeoffs have architectural implications, so your definition of success needs to be somewhat flexible—you may have to compromise on one design goal in order to achieve another. **How can you understand and monitor those system qualities that need to be addressed in a way that allows you to make thoughtful design tradeoffs?**

❖ ❖ ❖

It is important to identify essential qualities early so that they can be prioritized and contribute to your definition of done. You also need some flexibility in defining what's "good enough." So, you don't want hard and fast (inflexible) acceptance criteria values for all system qualities.

For some system qualities, there isn't one specific number you are aiming for, but you know what is minimally acceptable. For other qualities, you may have specific targets, but you are willing to compromise on them in order to achieve other system quality objectives. You want flexibility in achieving some quality requirements and overall accountability.

❖ ❖ ❖

Therefore, define and use an agile landing zone. A landing zone [Gilb] is a set of criteria used to monitor and characterize the "releasability" of a product. An *agile* landing zone [W2011a] is one where all the criteria and acceptable values are not fixed or known at the beginning. The criteria and values of an agile landing zone take shape over the lifetime of a project. Landing zone criteria are similar to release criteria, except they provide for tolerances in acceptable values. There isn't one number you are aiming for; you have a range of values for each system quality attribute you are targeting. This gives you some flexibility in defining what's "good enough," allowing you to make tradeoffs as long as you *Agree on Quality Targets.*

There are three possible values for any landing zone criteria: minimum, target, and outstanding. A minimum value is something you are willing to live with, although you may

aspire for a higher value. A target value is what you think you can achieve with reasonable cost and effort. An outstanding value is something that you believe might be achievable but not without significant effort. Sometimes minimum and target values may be identical; that just indicates that you have limited flexibility in achieving acceptable qualities.

Alternatively, where you have the least flexibility in your requirements, you might simply want to define acceptance criteria with specific values that must be met. Only use landing zones for those quality attributes that have some degree of flexibility in their outcome.

Table 1 is an example of a landing zone for a loan processing system (all the values have been concocted, for simplicity's sake; any relation to landing zones for real projects is coincidental). Each row represents a task that needs to be performed using the loan processing system.

The values represent the actual time it takes to complete a business task using the system. Tasks may or may not be initiated by users. Some are triggered by incoming data or by a change in the state of a loan (adjusting a loan's interest rate or assigning a loan processor). Others involve adding configuration data, writing additional code and deploying changes to production.

For example, a minimally acceptable time for the quality attribute "Adding a new loan agreement" is two weeks; the target is to enable the user enter all the information for a new loan agreement and have it configured into the system within 24 hours. "Adding a new loan product" is targeted for two weeks because several activities and actions that need to be completed in order to support processing a new kind of loan.

Quality Attribute	Minimum	Target	Outstanding
Adding new loan agreement	2 weeks	24 hours	12 hours
Add new loan product	3 weeks	2 weeks	1 week
Adjust loan	4 days	2 days	1 day
Access loan risk	1 day	6 hours	10 minutes
Assign loan processor	1 month	1 week	1 day

Table 1: Loan Processing System Landing Zone

Each row in the landing zone represents a measurable requirement (see *Measurable System Qualities*) which has a range of acceptable values labeled Minimum, Target, and Outstanding. The goal is to have each requirement within this range at the end of development. Inside the range is the desired value, labeled *Target*. *Minimum*, *Target*, and *Outstanding* are relative to your budget and timeframe.

If you have more than a few attributes, it can be helpful to organize your landing zone according to system quality category: e.g. performance, data quality, reliability, usability, etc.) and their priority. Table 2 illustrates a portion of a landing zone which has been organized according to the system quality category being measured.

Category	Quality Attribute	Minimum	Target	Outstanding
Performance	Throughput (transactions per day)	50,000	70,000	90,000
	Average transaction time	2 seconds	1 second	<1 second
	…			
Data Quality	Inter system data consistency (percent critical data attributes consistent)	95%	97%	98%
	Data Accuracy	97%	99%	>99%
	…			

Table 2: Landing Zone organized by Quality Category

A landing zone helps you focus on a few critical things to monitor (instead of hundreds). Your goal should be to only include those criteria that are critical to your project's success. If you do, it will be easier to see a bigger picture and make sense of it: when one attribute is edging below its minimum, what is happening with the others? Are they trending below minimum, too? If so, you have a big problem with achieving your overall product goals. No, and you have a landing zone which allows you to achieve a successful product/system launch even if every requirement isn't exactly on target.

Expect the criteria in an agile landing zone to shift and be adjusted over time. Initially, you may define those parts of your landing zone that you expect to achieve over the next few months, leaving the rest of the landing zone purposefully sketchy. What initially appeared to be achievable or reasonable targets may change in light of new facts or market changes. No one wants to deliver yesterday's product to today's market. Landing zones, like release criteria can and do change.

For example, you may have worked hard to meet some early achieved landing zone targets, only to find out that your early decisions had negative consequences on future work. You may have created some technical debt that either needs to be paid off in order to achieve your next targets. Given time or budget constraints, you may decide to *Recalibrate the Landing Zone* (and set expectations lower).

If you *Qualify the Roadmap* and include these system quality attributes in your *Agile Landing Zone* you can get a sense of when they should be considered. *Agile Landing Zone* targets can be made more visible to the team through various means such as putting them on *Quality Charts* or on the *System Quality Radiator*.

Measurable System Qualities

"Every line is the perfect length if you don't measure it." —Marty Rubin

To know whether a desired quality has been achieved it has to be measured. The description of the quality and the specific aspect you are trying to measure can't be vague or fuzzy. **How can you decide on what values you expect for a quality and how to measure them?**

❖ ❖ ❖

For system qualities, like performance or throughput, this may be relatively easy: performance can be measured by profiling system performance for a particular scenario, perhaps repeatedly to obtain an overall average.

Other qualities, like reliability, may require a complicated set of measures made over a period of time.

Some qualities, like usability, at first glance may appear entirely subjective and as a consequence impossibly difficult to measure. High-level quality attributes may need to be decomposed into smaller ones that are measured and aggregated.

Some qualities are difficult or costly to measure. Complex qualities can take quite a bit of effort to measure.

You want to make frequent measurements as you are designing and building your system so that you can react to changes in quality.

Balancing the time and effort required to make a measurement with the information it yields can be difficult.

❖ ❖ ❖

Therefore, define an appropriate way to measure a quality and to describe it with only as much accuracy and precision as you need. This involves defining or finding an appropriate way to measure (the meter) and describing accurately the values you expect (the scale) [Gilb].

There are three types of scales of measure: natural, constructed, or proxy. A natural scale is one that is obviously associated with a specific quality and is usually the easiest to agree upon. Examples are elapsed time to perform a system operation in milliseconds or the number of page hits in 24 hours. A constructed scale is one that is built specifically to measure a quality, for example, a 7-point user satisfaction scale.

Sometimes it is difficult to know how to directly measure system qualities. In this case you can use a proxy scale to measure parts of the system to give a feel for a certain system quality and then extrapolate expected values. A proxy scale is an indirect measure of quality, for example, projecting system throughput by using sample data and running transaction scripts.

Select a proxy scale if it would be too costly or time-consuming to measure a quality directly. It may also be that you need to construct a proxy scale when parts of the system are not yet completed or integrated. You may want to start by measuring using a proxy, then transition to a natural scale if you want to continue monitoring the quality in production.

Since adding necessary precision and accuracy can be difficult, especially for usability qualities, let's illustrate how to improve upon the extremely vague statement, "the system must be easy to use"

A first attempt adds more precision by identifying a specific task and what "easy" means for that task:

> *Eighty percent of novice users should successfully place an order for a single item in under 3 minutes without assistance.*

We can add more details; we're not only want to qualify the speed of placing on order but also whether online help is an aid or a hindrance:

> *Eighty percent of novice users should successfully place an order for a single item in under 3 minutes only using online help for assistance.*

There are two key ideas about measuring "easy to use." First, there is a scale which constrains the possible values of what we are measuring: Time required for a novice to complete a 1-item order using only online help for assistance. Second, there is a meter, which defines how we are going to make our measurement. Since we don't want to only measure one user and extrapolate to all users, we may decide on average the times obtained for 100 users during testing.

It's best to find a natural scale. People usually won't to argue about it being "good." If you can't find a natural scale, look next for a proxy. You may need to decompose what you are trying to measure into smaller parts and try again. For example, *"Adding a New Loan Agreement"* involves several sub-steps, each requiring time to perform. And you may need several different scenarios to specify expected values under different circumstances.

Finally, you may need to incorporate qualifiers to make things specific when you need the precision. For example, it isn't just any old user's response we're trying to measure, it is:

Time required for a novice to complete a 1-item order using only online help for assistance.

A meter can be an agreed upon way to provide a measurement. To find a meter, look at the scale. If no obvious meter comes to mind, ask others for their experiences or look for "off-the-shelf" tools that come with reasonable meters.

Once you've found a meter, check that:

- Stakeholders agree it is adequate,
- There isn't a more cost effective meter, and
- You can test it on the system, ideally, before it is deployed.

It is important to *Agree on Quality Targets* for whatever you measure. These *Measureable System Qualities* can ultimately be included in the *System Quality Dashboard*s that might be used to monitor the production system.

Agree on Quality Targets

"An agreement cannot be the result of an imposition." —Nestor Kirchner

There are several areas where you need to define specific quality-related targets. You may have targets for performance, usability, internationalization, reliability or other non-functional qualities that broadly apply to several user stories or across a number features. Or, you may have a specific system quality that you want to focus on and improve. How much improvement to strive for may be open to debate.

However, if you've done something similar in the past, the quality criteria to choose and their acceptable values may be obvious. At other times it can more of a challenge to reach agreement. **How can you reach consensus when defining quality acceptance criteria?**

❖ ❖ ❖

Diverse stakeholders have different interests, backgrounds and expectations. Not everyone may be equally informed. Some may hold contradictory opinions. Yet, in order to work towards a specific quality-related objective, everyone needs to buy in and work towards common measurable targets.

Technology constraints can limit what you can deliver at what cost. Sometimes technology choices are made due to business concerns or marketing trends. Technology decision have cost and quality ramifications that need to be considered.

Quality requirements priorities are often influenced by the effort to implement them and the effort needed to perform the benchmark.

❖ ❖ ❖

Therefore, work towards informed consensus on quality-related targets. Ideally a small group of informed individuals should agree upon target values. If you have diverse stakeholders with varying opinions, you may decide to give each stakeholder group a voice in identifying several qualities that are particularly relevant to them. These can be added to landing zone criteria that you've already established.

For first time landing zone builders, you might want to choose someone who knows about the product to take a first cut at establishing landing zone criteria [W2011b]. A business architect, product owner, or lead engineer might prepare a "proposed landing zone" containing reasonable values for quality criteria and values that are questioned, challenged, and then reviewed by a small group. For a landing zone, minimum, target and outstanding values should be agreed upon by the group. It is important to recognize how technical

considerations impact quality targets. Any assumptions about how these values can be achieved should be noted.

When you are coming up with specific values for quality scenarios, you might also use a similar approach. Some informed individual might make a rough cut at "proposed" values that are to be achieved. But a group of informed experts might refine initial values.

Discussions should be to the point, collaborative, and non-confrontational. Someone might propose a set of values based on historical trends and extrapolation. Or a software architect might propose values based on prototyping results or benchmark data. Or the team might declare a design spike to investigate reasonable and possible values. The group might end up agreeing to adjust numbers because the prototyping or design spike evidence was compelling. To effectively set quality value targets, the group should have mutual respect, trust and transparency, and no hidden agendas.

For example, on one program, the chief business architect made the initial cut of quality criteria and their initial values in the landing zone. He was a former techno geek who knew his technical limits. He had deep business knowledge, product vision and a sense about where to be precise and where there should be a lot of flexibility in the landing zone values. Consequently some criteria were very precise. Since they were in the business of processing a lot of transactions, they knew where they needed to improve based on projected increases in transaction volumes. The transaction throughput target for one business process was based on extrapolations from the existing implementation, taking into account the new architecture and system deployment capabilities. The minimum acceptable value was better than the current implementation (because why else would they be building a new system), but target and outstanding values were based on extrapolations of current capabilities. Other landing zone criteria related to maintainability were only generally categorized as requiring either a patch, a new system release, or online update support. The definitions for what was a patch, a release or an online update were nailed down so that there was no ambiguity in what was meant.

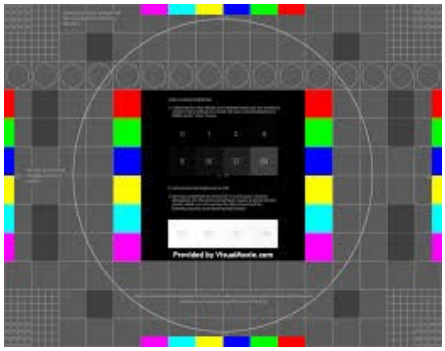
Possible ways of coming up with values include averaging informed individuals' estimates, using an existing system as a baseline, extrapolating values from similar scenarios, or benchmarking working code. Sometimes it may be necessary to create a spike solution to obtain estimates.

To reach consensus on specific quality scenario targets, you may need someone to play the role of facilitator. The facilitator should know enough about the program or product to be constructive, but they need not be the "authority" or "expert." That person should be good at gaining consensus and get the best from individuals who may have strongly held opinions and disagreements. Ideally, a facilitator knows enough about the product to offer constructive observations and has the ability to lead a small group forward in defining acceptable criteria and values. It can be more effective to have an informed facilitator to guide quality target definitions, than a dispassionate, uniformed one.

There are several times that an agile team needs to *Agree on Quality Targets*. For example, when an agile team is *Finding Essential Qualities* initial values for *Quality Scenarios* or *Agile Landing Zone* criteria need to be established. When *Recalibrating the Landing Zone,* attributes are modified and agreement should be made by an informed consent by a small group of people on changes to values.

Recalibrate the Landing Zone

"*Test fast, fail fast, adjust fast.*"— Tom Peters



Initially, you defined a set of landing zone criteria that you expected to achieve over a few iterations. You left the rest of your landing zone purposefully sketchy. As you've implemented new functionality, you have continued to add new landing zone criteria while monitoring the values of existing ones. **How can you continue to evolve your landing zone and keep it up to date?**

❖ ❖ ❖

As you continue with development, it can become harder to keep criteria within their landing zone target values. Solutions that achieve newly identified landing zone criteria may impact your ability to maintain other values.

What initially appeared to be achievable targets may change in light of new information and your current implementation.

It is important to not be constantly shifting back and forth on the targets. However, in the spirit of agile, as you learn new information the target values can be reconsidered and prioritized as needed.

Although the product owner or client will see these targets as important, they maybe see these as a lower priority over other things that need to be done and may not understand the implication to the overall system.

Budget and time constraints can limit the effort you are able to devote to achieving important quality constraints.

❖ ❖ ❖

Therefore, rather than simply throw up your hands in defeat, revisit your landing zone criteria and reset expectations. Some values may not be appropriate, given what you know now.

Because you are implementing your system incrementally and learning more about your system's capabilities and limitations, it is natural for the criteria in an *Agile Landing Zone* and their values to shift and be adjusted over time. What initially appeared to be reasonable targets will change in light of new facts or market changes. No one wants to deliver yesterday's product to today's market. Previous implementation decisions can affect or limit your ability to achieve newly identified criteria. For example, deciding to focus on meeting specific performance targets may have impacted usability or flexibility criteria.

Landing zones, like release criteria can and do change. In fact changing acceptable values for your landing criteria is not always a bad thing to do, especially if you are reacting to the current situation and making thoughtful design tradeoffs. This is part of your ongoing development cycle. It is important to prioritize work on quality targets and to maintain a balance between delivery of qualities and features.

For example, you may have created some technical debt that needs to be paid off in order to achieve some landing zone performance targets. Given time and budget constraints, you decide not to invest in design rework for the current release. It is more important to deliver working functionality on time than to make it fast. So, you opt to recalibrate your landing zone (setting acceptable performance criteria lower). You've made a conscious decision to redefine what is acceptable.

You might also recalibrate/readjust landing zone criteria upwards based on new information/system capabilities/technologies. For example, with experimentation you find that by tweaking cache and buffer sizes, you can increase throughput for an important data translation (ETL) process. Rather than simply move into the "outstanding" range, you also adjust the minimum acceptable value upwards and note that cache and buffer tuning should be considered for any time critical ETL process. When a team is recalibrating the landing zone it is often the case that the team will need to *Agree on Quality Targets*.

System Quality Dashboards

"The dashboard needs to deliver data in a timely fashion, and that timeliness is dictated by which process is being represented in the dashboard" —Keith Gile

Typically, agile software development focuses on features and functionality first before paying attention to other important system aspects such as architecture and critical qualities. On agile projects you hear statements like, "Make it work, make it right, then optimize it." Most agile practices push to develop important functional requirements as outlined by the product owner, which are prioritized on the work backlog. As the system evolves the team begins to better understand what system qualities are important and how to better measure them. As the system evolves, keeping track of these qualities becomes increasingly important.

How can agile teams provide a means to make this information accessible and visible to the team?

❖ ❖ ❖

Creating tools and dashboards takes time and often there are limited resources and people dedicated to building QA tools. Tools and dashboards can seem like a pointless luxury compared to making sure the system is meeting the requirements well enough to ship.

It can be difficult to know what qualities are important to monitor. As more and more qualities are built into the system, some are important to keep a watch on while others, once validated and made testable, are good enough.

If certain qualities such as security, performance and reliability are not regularly tracked, they can be difficult to improve late in the development process. Although originally the system might meet quality constraints, as the system evolves, qualities can degrade if they aren't monitored and maintained.

Some qualities can be hard to accurately measure until the system functionality is complete. However, you want to know as the system evolves, whether you can achieve system quality goals.

❖ ❖ ❖

Therefore, create dashboards to test and validate important qualities. As important system qualities are outlined and included in the backlog, note which ones should be monitored and where tools can be created to measure the system as it evolves.

The first step is to outline the critical items that need to be measured and monitored on an ongoing basis. Some of these can start off with simple measures, exercising and measuring only partially implemented functionality. Initially, these can be incorporated into your existing test framework. Although initially you might be making simple measures, unless you incorporate them into a dashboard, they won't be readily visible. So you might want to

incorporate these measurements into a dashboard that provides ongoing feedback on important qualities as the system evolves. This is a form of "*Continuous Inspection*" [MYGA].

When should you build a dashboard or when should you buy one? When selecting a tool, it is important to know how easy it is to set up; i.e. does it "*Work Out of the Box*"[FY98]. If a tool provides all that is needed and is relatively easy to use, use the tool. However, some tools that provide powerful means of measurement can be costly or hard to use. So you need to decide between purchasing a powerful tool or using an open source dashboard that may not be as powerful. Another consideration is how well the dashboard integrates with your development environment.

Whether you buy or build a tool, consider what quality aspects should be shown and what frequency that they are measured. Does the dashboard perform quality-related tests when initiated by a user, showing results of tests executed during build or integration, or is it monitoring the production system? How frequently should contents of a dashboard be updated in order to be useful and what happens when measured values fall below minimally acceptable criteria? Some dashboard tools allow you to configure alerts and notifications when measured values cross a threshold. Figure 1 is an example of some third party open source tools for monitoring systems such as SonarCube.

Figure 1: Quality Dashboards

Dashboards can show real-time results, for example, performance of running processes, or display quality values measured during check-in or system build quality tests. These dashboards can overlap with operational dashboards for production systems.

As the team *Recalibrates the Landing Zone* it is important to refine dashboards to include the newly updated values.

Summary

This paper extends a set of initial patterns about "Becoming More Agile at Quality." The complete set of patterns includes both ways of incorporating QA into an agile process and agile techniques for describing, measuring, adjusting, and validating important system qualities. The patterns in this paper are related to measuring and monitoring qualities. Ultimately the authors intend to write all of the patlets in the appendix as patterns and weave them into a 3.0 pattern language for evolving and embedding Quality Assurance into an Agile Quality mindset.

Acknowledgements

We thank our shepherd Eduardo Guerra for his valuable comments and feedback during the PLoP 2014 shepherding process. We also thank our 2014 PLoP Writers Workshop Group, Yasunobu Kawaguchi, Michael John, Yu Chin Cheng, Hironori Washizaki, Marvin Toll, and Eduardo Guerra for their valuable comments.

References

[GILB] Gilb, Tom. 2005. *Competitive Engineering: A Handbook For Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage.* Butterworth-Heinemann.

[FY98] Foote B., and Yoder, J. 1998. *The Selfish Class*. Proceedings of the 3rd Conference on Pattern Languages of Programs (Monticello, Illinois). PLoP '96, Technical Report #WUCS-97-07, September 1996, Department of Computer Science, Washington University. Pattern Languages of Program Design 3 edited by Robert Martin, Dirk Riehle, and Frank Buschmann, Addison-Wesley, 1998

[IBA] Iba, T. 2011. "Pattern Language 3.0 Methodological Advances in Sharing Design Knowledge," International Conference on Collaborative Innovation Networks 2011 (COINs2011).

[MYGA] Merson, P, Yoder J., Guerra E., and Aguilar A., "Continuous Inspection: A Pattern for Keeping your Code Healthy and Aligned to the Architecture," 3rd Asian Conference on Patterns of Programming Languages (AsianPLoP 2014), Tokyo, Japan, 2014.

[YWA] Yoder J., Wirfs-Brock R., and Aguilar A., "QA to AQ: Patterns about transitioning from Quality Assurance to Agile Quality," 3rd Asian Conference on Patterns of Programming Languages (AsianPLoP 2014), Tokyo, Japan, 2014.

[W2011a] Wirfs-Brock R., (July 28, 2011). *Agile Landing Zones,* http://wirfs-brock.com/blog/2011/07/28/agile-landing-zones/

[W2011b] Wirfs-Brock R., (August 16, 2011). *Who Defines (or Redefines) Landing Zone Criteria,* http://wirfs-brock.com/blog/2011/08/16/who-defines-or-redefines-landing-zone-criteria/

Appendix

A previous paper on this topic outlines some core patterns when evolving from traditional quality assurance to being agile at quality [YWA]. We outlined the patterns using patlets. A patlet is a brief description of a pattern, usually one or two sentences. Following is an excerpt from that paper outlining the patlets.

> We break our software-related Agile Quality patterns into these categories: fitting quality into your process, identifying system qualities, making qualities visible, and being agile at quality assurance. This paper will outline twenty-four patlets organized into four categories: knowing where quality concerns fit into your process, identifying system qualities, making quality visible, and being agile at quality assurance. We expect to evolve and extend these categories and patterns over time.

Our ultimate goal is to turn all patlets into full-fledged patterns and make a pattern language for action and change useful to software teams who want to become more agile about system quality.

Core Patterns

Central to using these QA patterns is breaking down barriers and knowing where quality concerns fit into your agile process. The following patlets describes these considerations.

Patlet Name	**Description**
Break Down Barriers	Tear down the barriers between QA and the rest of the development team. Work towards engaging everyone in the quality process.
Integrate Quality	Incorporate QA into your process including a lightweight means for describing and understanding system qualities.

Identifying Qualities

An important but difficult task for software development teams is to identify the important qualities (non-functional requirements) for a system. Quite often system qualities are overlooked or simplified until late in the development process, thus causing time delays due to extensive refactoring and rework of the software design required to correct quality flaws. It is important in agile teams to identify essential qualities and make those qualities visible to the team. The following patlets support identifying the qualities:

Patlet Name	**Description**
Find Essential Qualities	Brainstorm the important qualities that need to be considered and list them for inclusion on the product roadmap.
Agile Quality Scenarios	Create high-level quality scenarios to examine and understand the important qualities of the system.
Quality Stories	Create stories that specifically focus on some measurable quality of the system that must be achieved.

Measurable System Qualities	Specify scale, meter, and values for specific system qualities.
Fold-out Qualities	Define specific quality criteria and attach it to a user story when specific, measurable qualities are required for that specific functionality.
Agile Landing Zone	Define a "landing zone" that defines acceptance criteria values for important system qualities. Unlike traditional "landing zones," an agile landing zone is expected to evolve during product development.
Recalibrate the Landing Zone	Readjust landing zone values based on ongoing measurements and benchmarks.
Agree on Quality Targets	Define landing zone criteria for quality attributes that specify a range of acceptable values: minimally acceptable, target and outstanding. This range allows developers to make tradeoffs to meet overall system quality goals.

Making Qualities Visible

It is important for team members to know important qualities and have them presented so that the team is aware of them. The following patlets outline ways to make qualities visible:

Patlet Name	**Description**
System Quality Dashboard	Define a dashboard that visually integrates and organizes information about the current state of the system's qualities that are being monitored.
System Quality Radiator	Post a display that people can see as they work or walk by that shows information about system qualities and their current status without having to ask anyone a question. This display might show current landing zone values, quality stories on the current sprint or quality measures that the team is focused on.
Qualify the Roadmap	Examine a product feature roadmap to plan for when system qualities should be delivered.
Qualify the Backlog	Create quality scenarios that can be prioritized on a backlog for possible inclusion during sprints.
Quality Chart	Create a chart or listing of the important qualities of the system and make them visible to the team; possibly on the agile board.

Being Agile at Quality

In any complex system, there are many different types of testing and monitoring, specifically when testing for system quality attributes. QA can play an important role in this effort. The role of QA in an Agile Quality team includes: 1) championing the product and the customer/user, 2) specializing in performance, load and other non-functional requirements, 3) focusing quality efforts (make them visible), and 4) assisting with testing and validation of quality attributes. The following patlets support "Becoming Agile at Quality":

Patlet Name	**Description**
Whole Team	Involve QA early on and make QA part of the whole team.
Quality Focused Sprints	Focus on your software's non-functional qualities by devoting a sprint to measuring and improving one or more of your system's qualities.
QA Product Champion	QA works from the start understanding the customer requirements. A QA person will collaborate closely with the Product owner pointing out important Qualities that can be included in the product backlog and also work to make these qualities visible and explicit to team members.
Agile Quality Specialist	QA provides experience to agile teams by outlining and creating specific test strategies for validating and monitoring important system qualities.
Monitor Qualities	QA specifies ways to monitor and validate system qualities.
Agile QA Tester	QA works closely with developers to define acceptance criteria and tests that validate these, including defining quality scenarios and tests for validating these scenarios.
Spread the Quality Workload	Rebalance quality efforts by involving more than just those who are in QA work on quality-related tasks. Another way to spread the work on quality is to include quality-related tasks throughout the project and not just at the end of the project.
Shadow the Quality Expert	Spread expertise about how to think about system qualities or implement quality-related tests and quality-conscious code by having another person spend time working with someone who is highly skilled and knowledgeable about quality assurance on key tasks.
Pair with a Quality Advocate	Have a developer work directly with quality assurance to complete a quality related task that involves programming.