Quality Assumption Reviews

When developer understanding of quality issues is shallow and uneven, it may help to hold an assumptions review. The review synchronizes stakeholder understanding of quality goals, how they differ from functional requirements, and how they are achieved. It is an effective way to prepare to identify the project's quality goals. It can be the first step in building a culture of quality awareness.

To prepare for a review, list project-specific assumptions about qualities, their relationships, their achievement, and their verification. Your list can be supplemented by tailoring the generic quality assumptions that follow to reflect project specifics. Other assumptions may be added during the review session.

The list is given to stakeholders before the review session. Assumption discussions before the review session should be encouraged.

During the session, stakeholders discuss the assumptions and asks questions to clarify them and explore their project impact. Stakeholders identify problems with the assumptions by providing counterexamples or other reasons.

Open issues and unanswered questions are recorded during the review session. Resolutions should be completed and communicated within two weeks following the session.

If your organization does not provide a "culture of safety", many stakeholders will not engage during the review. If the review group is "too large" (e.g. > 9), stakeholder engagement may also be a problem. Multiple review meeting may be needed.

List of Generic Quality Assumptions

1. Common misconceptions about qualities
   - Early identification of quality goals is unnecessary
   - Qualities emerge during a project, like functions
   - Quality goals can be expressed as user stories
   - Developers understand and effectively achieve quality goals
   - Testing is enough to verify qualities
   - Coding standards are optional

2. Many quality goals are much riskier than most functional goals. [Consider the frequency and volume of recent hacks.]

3. Quality goals, their relationships, and attributes are **invariant** across application domains and should be described in a **rich quality model template**. Relevance, priorities, levels, and implementations must be determined while developing the **project quality model**.

4. Most quality goals (e.g. understandability, safety, throughput, scalability) **crosscut functions** and thus can't be implemented in a single increment and have an increasing cost to implement or repair across iterations.

   For example, "each custom module must comply with our coding standards" and "each custom module must implement exception handling" are tactics that support the crosscutting goals of understandability and robustness. When a developer creates a functional module e.g. to make a reservation, they must also be aware of all associated tactics supporting crosscutting goals e.g. complying with coding standards, otherwise they create technical debt.

5. Some quality goals entail **levels of achievement** e.g. security, privacy, and performance. For example, if security is relevant, should it be **minimal** using a user name and password, **moderate** using user name, password, and security questions, or **strong** using a retinal scanner?

6. Understanding quality goals entails understanding their **feasibility** (i.e. achievability). Is 99.999% reliability possible in your situation?

7. Some pairs of qualities **conflict** e.g. understandability and performance, while others **support** e.g. security supports privacy

8. Some qualities are **essential for all applications** e.g., (1) compliance with required rules, (2) domain sufficiency, (3) understandability, and (4) verifiability.

9. Some qualities are **universal** in the sense that it is hard to imagine a system where they don't matter. These include the essential qualities (assumption 8) as well as: ease of use and learning, modularity, reliability, and responsiveness.

10. Quality goals are achieved using tactics e.g. supports and self-checking results. There are at least four types of supports; (1) other qualities (2) system functions e.g. logging & exception handling (3) sets of rules e.g. coding standards & design patterns and (4) warning labels e.g. "are you sure that …".

11. Many quality goals are **harder to understand, express, achieve, and verify** than most functional goals. Even when the quality goal is precise and you know the candidate supports, you must select the specific supports needed to achieve the goal based on the system-specific execution environment.
12. Some quality goals (e.g. availability, performance, privacy, reliability, safety, security, and usability) are **much harder to understand** than other quality goals. This is because each is a software engineering subfield with an extensive body of knowledge.
13. Verifying quality goals is **much more difficult** than verifying functional goals. Quality goals, levels, and tactics must be verified with composite strategies that include technical review, analysis, measurement, and test. Testing alone is inadequate.
14. Effective verification of some quality goals (e.g., security, safety, robustness, reliability, recoverability, modifiability, reusability, and extensibility) requires accurate anticipation of future events.
15. Quality goals should be **monitored** with built-in tests, exception logging, and service call tracking.
16. Understanding one quality goal does not help in understanding another. Each goal (e.g. safety, reliability, and performance) has its own set of concepts that must be understood to achieve the goal.
17. Quality goals and functional goals have **little in common** [See comparison below].
18. Customers focus on functions, but quickly notice the absence of quality.
19. Implementation details about instantiated tactics need to be **communicated to developers**. In addition, project learnings about quality goals need to be recorded. Some form of documentation is necessary. Quality model templates and coding standards can be used.
20. Need to identify quality goals, levels, and supports ASAP to help minimize technical debt
    a. Identify relevant quality goals by traversing a rich quality model template
    b. Identify quality levels using goal scenarios e.g. what level of security?
    c. Identify other qualities, system functions, rules, and warning labels that support the identified quality goals and levels
    d. Assess adequacy of supports for each quality goal
    e. Resolve quality and support conflicts via avoidance or prioritization
    f. Develop verification strategies for quality goal achievement including measures e.g. measures for ease of use
    g. Develop a master list of quality supports
21. Having a single “quality expert”, without access to others, is a risky staffing strategy because no one is competent to review their own work.
22. “Quality experts” need to share quality understanding with the team e.g. using assumption reviews and brown bag lunches.
23. A team needs to understand its (quality) limitations and how to compensate for them.

Comparison of quality goals and functional goals

Qualities	Functions
• are rarely part of the rationale for the application	• are the primary rationale for the application
• are detailed by architects, designers, or developers	• are elicited from customers and application and domain experts
• are pervasive properties	• are local properties
• affect the choice of architecture or the implementation of multiple domain functions i.e. pervade the code	• affect individual modules i.e. are localized in the code
• compliance can occur at different levels e.g. high, medium or low security, or be binary i.e. yes or no	• compliance is binary
• can only be fully demonstrated within later iterations	• can be fully demonstrated within any iteration
• require analysis, technical reviews, and measurement along with testing for adequate verification	• can often be adequately verified with thorough testing alone
• change infrequently	• may change frequently
• may not be changed at a reasonable cost late in the project	• may be changed at a reasonable cost late in the project
• are achieved using a set of supports that are quality and degree specific	• can be decomposed into subfunctions
• may support or conflict with one another	• are often independent of one another
• implementation may entail reuse of system function modules (from libraries) within and across iterations e.g. reuse of exception handling module	• reuse is limited to similar applications
• cost of delayed identification can be significant due to (1) lack of consideration in architecture decisions, (2) lack of time to learn about supports, (3) cost of regression testing when supports are added, and (4) incremental delivery of software that may have valuable functions, but is missing valuable qualities	• cost of delayed identification is often small
• delayed implementation has increasing delay cost across iterations	• delayed implementation has a fixed delay cost across iterations
• estimates of quality support costs should increase in accuracy with increased experience across application domains	• estimates of functional costs may not increase in accuracy with increased experience across application domains