Agile is a Quality Anti-Pattern

David Gelperin
ClearSpecs Enterprises
Golden Valley, MN
david@clearspecs.com

***Abstract*— This paper identifies problems with the Agile approach to quality goals and describes solutions for Agile and all development methodologies.**

Keywords— ***Agile; quality goals; problematic principles; missing supports; quality-aware development***

I. Introduction

A. Quality Attributes

There are over 50 software quality attributes (ilities) including security, safety, and robustness. Each quality attribute has over 20 characteristics including priority, conflicting qualities, dependencies, and support strategies including those for achievement and verification.

Many (over 30) of these attributes are supported by many (over 12) other "basic" attributes. A few attributes (safety, error resistance, reusability, portability, and dependability) are supported by the basic attributes plus many (over 12) other attributes i.e. by more than two dozen attributes. This means that if safety is a necessary quality attribute, then more than two dozen other attributes are required to support it (Figure 1).

These things are true across all applications and all domains.

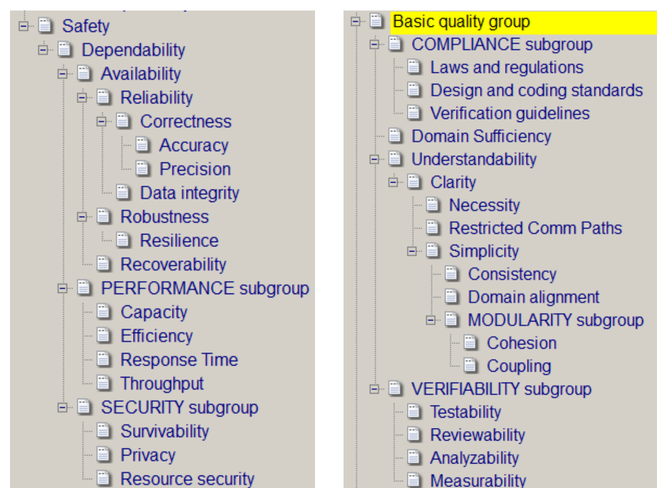

Fig. 1. Fragments of a quality goal taxonomy showing support for safety.

Information about quality attributes is scattered and education is meager. There is only one comprehensive, detailed, and succinct overview [1]. University courses provide in-depth coverage of a few attributes such as availability, performance, privacy, reliability, safety, security, and usability, but do not detail the others. In addition, there is little guidance in most software development organizations focused beyond the seven attributes previously mentioned.

As a result, many developers have an inadequate understanding of quality attributes and how to achieve and verify them. Worse, they don't know they don't know. This lack of understanding endangers software quality and project success.

B. Quality Goals

We refer to "required quality attributes" as "quality goals". Each quality goal needs to be defined, have its feasibility assessed, and then be achieved, and verified. To assess feasibility, a quality goal's achievability, verifiability, and cost consequences must be thoroughly understood. Defining a feasible collection of quality goals can be difficult because goals may conflict or their total cost may be unacceptable.

Defining a collection of functional requirements is like designing a mural. Defining a collection of quality goals is like designing a mobile. It's about finding the balance points.

C. Agile

Agile is a set of values and principles [2] for software development. It is not a development methodology. Agile methodologies support its values and principles. There are three types of methodologies:

1. **named** [3] including Extreme Programming (XP), Scrum, Crystal, Dynamic Systems Development Method (DSDM), Lean Development, and Feature-Driven Development (FDD);

2. **pure-hybrid** i.e. a blend of 2 or more named methodologies;

3. **mixed-hybrid** i.e. a blend of 1 or more named methodologies and non-Agile practices e.g. identifying most quality goals up-front.

All Agile methodologies involve continual evolution i.e. iterative, incremental development, driven by customers and evolving understanding i.e. change.

Big Requirements Up-Front (BRUF) is considered an Agile anti-pattern, because BRUF is inconsistent with evolving understanding caused by incremental development.

Note that BRUF is only inconsistent if requirements understanding evolves during development. If developers fully understand the requirements based on experience with similar systems, there is no inconsistency. Iterative, incremental development may still be a good idea, but not because of inadequate understanding of requirements.

II. Problem Definition

A. Agile Quality

Consider the quality guidance provided by two Agile methodologies, Scrum and XP.

Scrum [4], the most popular Agile methodology, provides no guidance on defining, achieving, or verifying quality goals.

XP [5] provides significant guidance on three quality attributes by detailing a set of practices. These include:

- pair programming and thorough code review and unit testing of all code
- test-first development i.e. planning and writing tests before each increment
- automated testing
- coding standards (not followed by 2/3 of Agile projects according to a 2010 survey [6])
- simple design
- refactoring

XP focuses on clean, understandable, and effective code and unit tests thus supporting three (of the more than 50) quality attributes: sufficient functionality, reliability, and understandability of code and unit tests. Since understandable and reliable code supports most other quality goals, XP provides necessary, but insufficient support for most other attributes.

A (limited) view of quality in named and pure hybrid Agile has been summarized [7] as follows:

"Quality is an inherent aspect of true agile software development. The majority of agilists take a test-driven approach to development where they write a unit test before they write the domain code to fulfill that unit test, with the end result being that they have a regression unit test suite at all times. They also consider acceptance tests as first-class requirements artifact, not only promoting regular stakeholder validation of their work but also their active inclusion in the modeling effort itself. Agilists refactor their source code and database schema to keep their work at the highest possible quality at all times."

B. Agile's problematic principles [8]

We now consider five of Agile's 12 principles.

- **2. Welcome changing requirements, even late in development**. Agile processes harness change for the customer's competitive advantage.

  Welcoming (rather than avoiding) changing crosscutting quality requirements late in development **is a bad idea**. Such expensive changes usually result from voluntary ignorance rather than the emergence of unimaginable quality goals. This principle can be fixed by adding "functional" to "requirements".

- **6. The most efficient and effective method of conveying information** to and within a development team **is face-to-face conversation**.

  Face-to-face conversation is great. It is NOT the most efficient nor effective method of conveying information about quality goal definitions nor about quality achievement and verification strategies. This principle can be fixed by adding "many kinds of" to "information".

- **11. The best** architectures, **requirements**, and designs **emerge from self-organizing teams**.

  Expecting "the best" quality requirements to emerge prior to delivery is a poor strategy. Quality requirements don't need to emerge because they can be selected early in a project from a quality knowledge base [1]. Emergence is great, when experience and understanding are lacking. It is inefficient and expensive, when the choices are known. This principle can be fixed by adding "functional" to "requirements".

- **3.** Deliver **working software** frequently, from a couple of weeks to a couple of months, with preference to the shorter timescale**.**

  7. Working software is the primary measure of progress.

  What is "working software"? Without identifying quality goals, developing achievement and verification strategies, and implementing crosscutting quality supports FIRST, the early increments can't be defect-free nor satisfy their quality requirements. These principles can be fixed by defining "working software" as "a possibly-fragile prototype sufficient to demonstrate the functionality to be delivered"

Agile is currently based on several anti-quality principles. These principles need to be fixed or deleted to improve Agile's quality support.

C. Agile's quality drawbacks

Named and pure-hybrid Agile's view of quality has an **extreme bottom-up functional bias** as suggested by these characteristics:

- Discourages specifications because code and tests are considered satisfactory;
- No specification or analysis of quality threats (e.g. safety and security) or of strategies for achievement and verification. Threats (e.g. hazards and attacks) and mitigation strategies are the key to understanding the effectiveness of crosscutting supports;
- Discourages up-front analysis and design for fear of waste, including gold-plating;
- Focus on testing, rather than verification;
- Emphasis on incremental design, which is ineffective for crosscutting supports;
- No mention of risk management, resolving quality conflicts, or designing crosscutting quality supports.

XP, done well, is wonderful at achieving functional goals and three quality attributes.

All named and pure-hybrid methodologies are terrible at achieving and verifying the other 50 quality attributes, because:

1. At best, Agile treats quality goals like functional goals. At worst, it ignores them. Quality goals "emerge" at unspecified times during a project, because up-front analysis is discouraged. Quality goals are often documented with user stories, put in a backlog, and selected for implementation when their priority forces them to the top.

2. Since customers rarely consider quality goals, unless prodded, nothing in Agile assures that all high-priority quality goals will emerge before product delivery.

3. Agile emphasizes functions and de-emphasizes most quality goals to the point of invisibility.

4. Agile emphasizes testing and, except for code and tests, de-emphasizes analysis, review, and measurement to the point of invisibility. Verifying quality goals requires technical activities other than testing e.g. development and review of hazard lists and verification strategies.

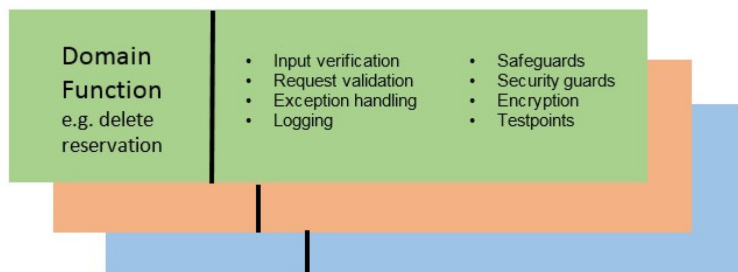

Fig. 2. Contents of software components

D. Most Agile methodologies cause reckless short-term technical debt [9]

Most functional components must contain code to support quality goals (Figure 2).

Developing a "working" component with some necessary quality supports missing often results in reckless short-term technical debt. Most Agile methodologies create such reckless debt since few begin by identifying quality goals.

This early ignorance of quality goals is **voluntary**. Most quality goals can be accurately identified from knowledge of the software's operating environments and basic mission e.g. flight control, internet gaming, or stock trading, and use of a quality knowledge base [1]. Early identifications may need to be adjusted as understanding deepens, but there is no way to predict when there is enough information to accurately determine a quality goal without waiting until all functional code has been written. Waiting is an expensive alternative to early identification.

Any development methodology that does not start by identifying quality goals and their achievement and verification strategies using a quality knowledge base is unnecessarily expensive and may produce quality-deficient software. Unfortunately, this includes most development methodologies, not just Agile ones.

At best, when early identification does not happen, all relevant quality goals emerge and all their adequate supports are achieved. However, there is a large cost for refactoring the reckless technical debt that results i.e. development is very inefficient. In addition, the adequacy and achievement of the quality goals is unknown before delivery.

At worst, when early identification does not happen, many relevant quality goals are missed, many necessary supports are missing or unreliable, and there is a large cost for refactoring the reckless technical debt that results i.e. development is ineffective and very inefficient.

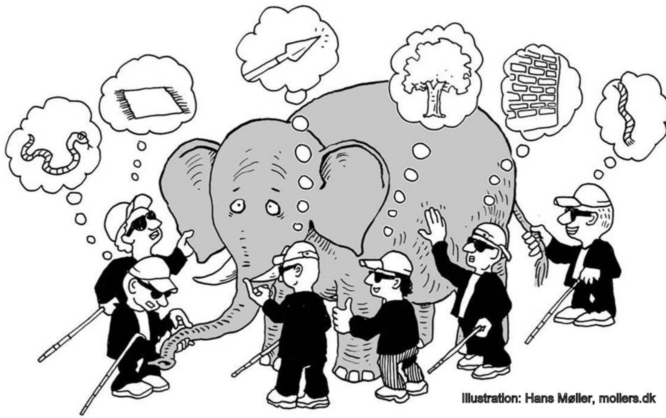

Fig. 3. The blind men and the elephant

We are living the parable of the blind men and the elephant (Figure 3) – detailed understanding of some quality goals (i.e. the software engineering subfields of availability, performance, privacy, reliability, safety, security, and usability) by a few, but little grasp of the entire set of goals i.e. how to achieve and verify the other attributes.

III. Proposed Solutions

The challenge is to raise stakeholder awareness of quality goals to the same level as their awareness of functional goals. Quality awareness implies early understanding of:

- high-priority quality goals and their characteristics
- conflicts between quality attributes and how they should be resolved
- critical supports for each quality level
- effects of the critical supports on each domain function
- how qualities will be verified

Quality awareness includes the use of **Quality-Aware development** [10]. Quality-Aware development is NOT a development methodology, but a 3-part supplement to whatever you are doing now or intend to do (including the use of Agile development). Quality-Aware Agile is a mixed-hybrid methodology.

The first part of the supplement is an initial quality sprint that includes:

1. Select relevant quality attributes including their supporting attributes from your quality knowledge base;
2. For each selected attribute: identify its required level, identify its challenges, mitigations, and supports, assess its feasibility and then specify and review its achievement and verification strategies;
3. Analyze each pair of potentially conflicting quality attributes to identify and resolve the real conflicts so that adequate architectures can be identified.

The second part is a set of tasks to be added to each development iteration. For each iteration-relevant quality goal:

1. Reassess its achievement and verification strategies and update as needed;
2. Carry out its achievement strategy, clearly identifying quality support code;
3. Verify its achievement and change as needed.

The third part is to collect quality learnings during a project retrospective and record them in the quality knowledge base and/or in the development standards.

More tactics for increasing quality awareness are described in [10]. A quality knowledge base and more resources are freely available [11].

We recommend Quality Goals First (QGF). Failure to practice QGF always results in reckless short-term technical debt. You can estimate the cost of this reckless debt by multiplying the total number of quality-incomplete components produced during development by the average cost of refactoring this kind of debt.

Identify **quality before functionality** to improve results. Don't worry about gold-plated quality. It is unlikely.

References

[1] D. Gelperin **LiteRM Quality Knowledge Base**, freely available [11]
[2] … http://www.agilemanifesto.org 2001
[3] … https://en.wikipedia.org/wiki/Agile_software_development
[4] K. Schwaber and J. Sutherland **Scrum Guide** 2013
[5] K. Beck **Extreme Programming Explained** Addison-Wesley; 2nd edition 2004
[6] S, Ambler http://www.ambysoft.com/surveys/howAgileAreYou2010.html 2010
[7] S. Ambler "Quality in an agile world" **Software Quality Professional** Vol. 7 No. 5 2005
[8] … http://www.agilemanifesto.org/principles.html 2001
[9] M. Fowler "Technical debt quadrant" blog post http://martinfowler.com/bliki/TechnicalDebtQuadrant.html 2009
[10] D. Gelperin "Failure is becoming the norm", freely available [11]
[11] … www.quality-aware.com