# Agile is a Quality Anti-Pattern (and what you should do about it)

David Gelperin, CTO
ClearSpecs Enterprises

I. Introduction
II. Problem Definition
III. Proposed Solution

# Disabusings

I like Agile – It is a powerful framework for dealing with (portions of) the software development challenge.

Most things have strengths and weaknesses – Agile is no exception.

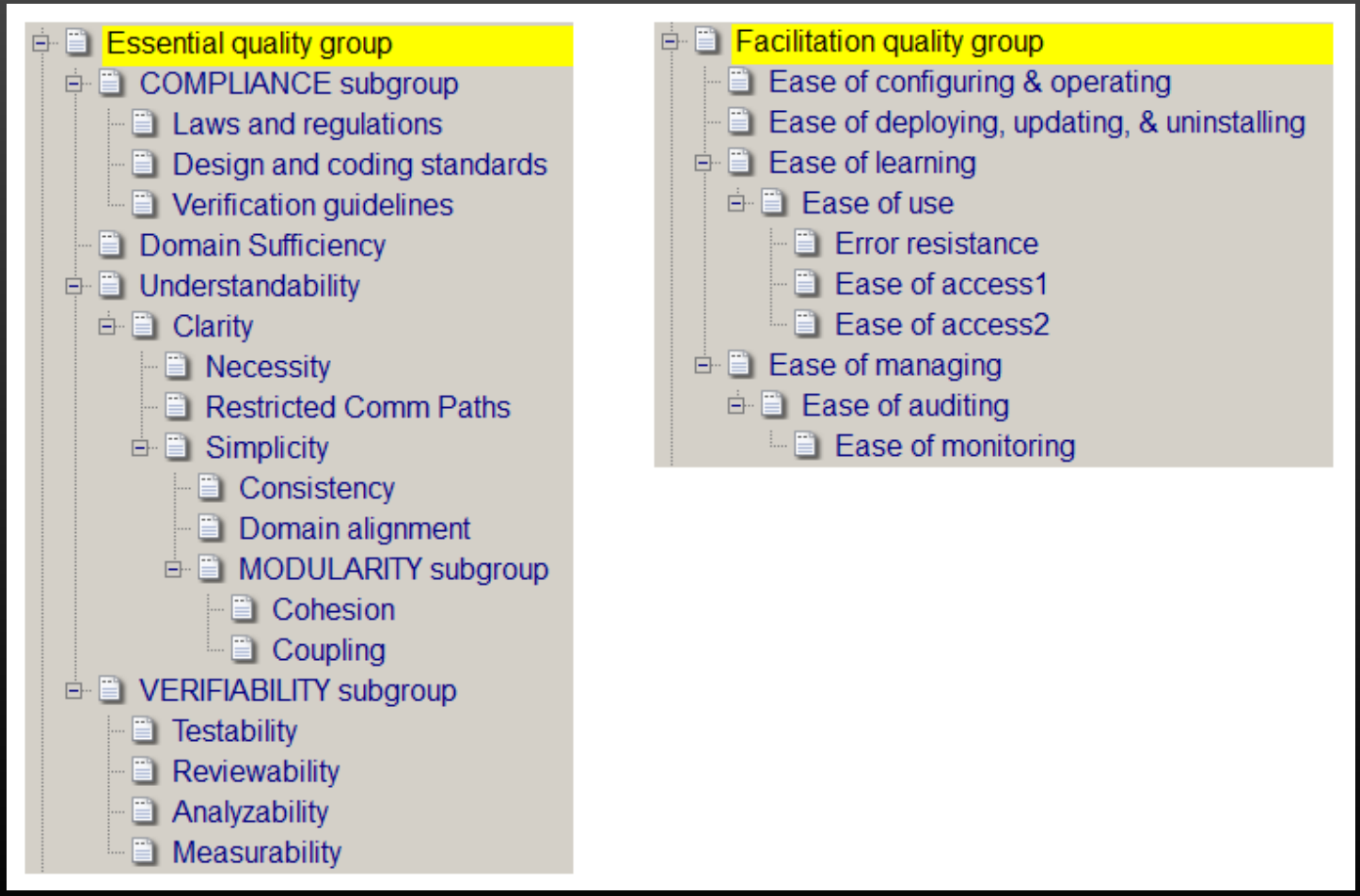
# Introduction -- Quality attributes

There are over 50 software quality attributes including survivability, safety, and robustness.

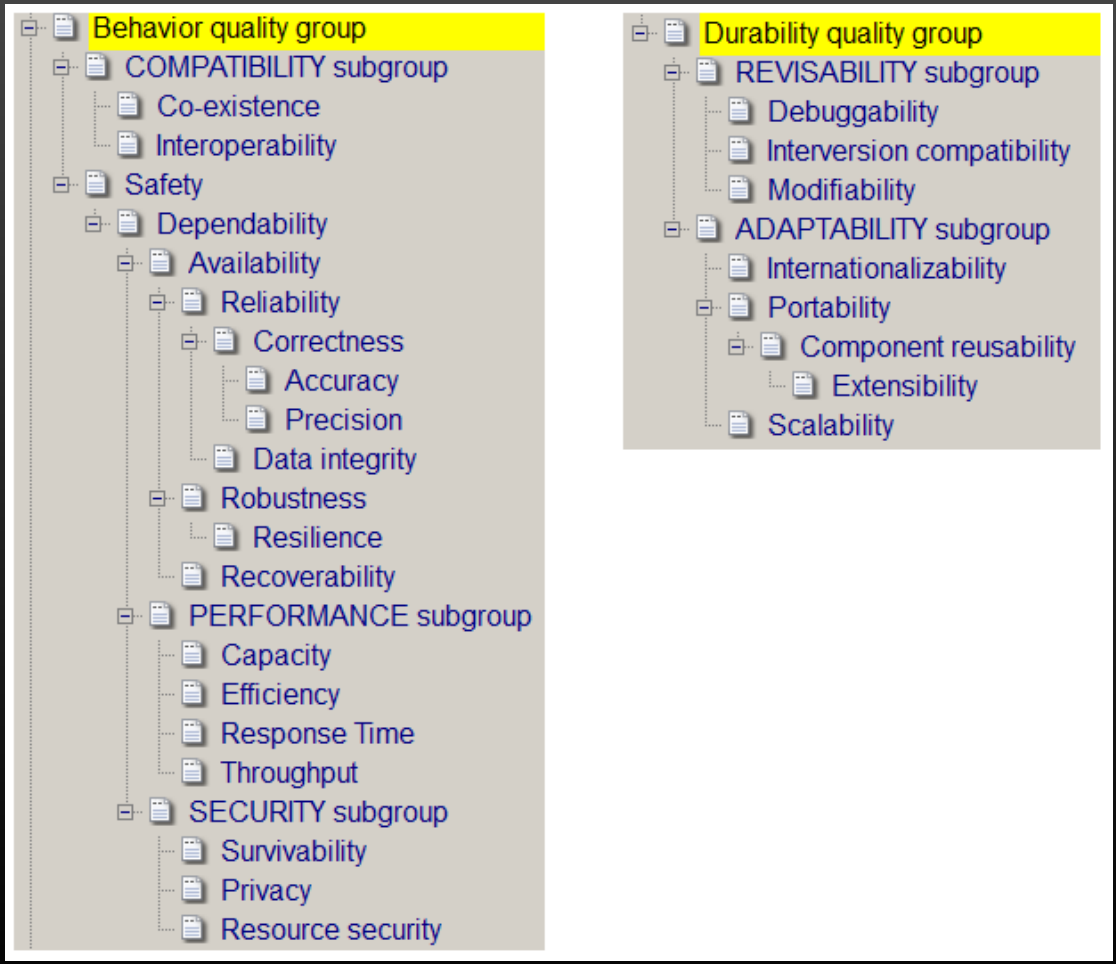
Each quality attribute has over 20 characteristics including priority, conflicting qualities, supporting qualities, and achievement and verification tactics.

This is true across all applications and all domains.

# Attribute examples -- 1 of 2



# Attribute examples -- 2 of 2



# Characteristic examples -- 1 of 3

**Reliability**

**Definition** Degree to which system effectively performs requested functions under all conditions

**Software subfield** reliability engineering

**Assumptions/Rationale**

**Indicators** [measures of supporting and directly supported attributes]

**Measures** Mean time to failure (trailing), Mean time between failures (trailing), Failure rate (trailing)

**Aspect of** availability, modifiability

**Supporting qualities** correctness, data integrity

**Associated with** Robustness, Recoverability, Modifiability, Component reusability

**Conflicting qualities** performance, many adaptation qualities

**Threats** Defective code or data, hardware failure, excessive customization

**Mitigations** Formal review, data analysis, thorough testing, reliability measures tracking

**Additional supports**

- Monitor and control system states and data integrity (e.g. output)
  - Specialized interfaces
- Comprehensive exception handling
- Maximize reuse of reliable components

# Characteristic examples -- 2 of 3

**Constraints** Design and coding standards that limit complexity, verification guidelines requiring technical reviews, measurement, analysis, and comprehensive usage, code, and data test coverage as well as exploratory testing

**Verification tactics** review standards compliance, review and test code including exception handlers, analyze data, measure and track mean time to failure, verify all supporting qualities

**Elicitation Questions**

- Which system aspects threaten reliability
- What is acceptable reliability
- Which functions require ultra-high reliability
- How will unreliability of inhouse and third-party software be detected and communicated
- How to handle unreliable external systems

**References**

**The Quest for Software Requirements** -- section 5.5

**Software Reliability Engineering**

**Automated Source Code Reliability Measure OMG/CISQ**

# Characteristic examples -- 3 of 3

**Risk Factors**

a. Developer understanding = [superficial, limited, deep]

b. Cost (implementation, verification, maintenance) = [high, medium, low]

c. Feasibility (technical, cost, understanding) = [low, medium, high]

**Other Characteristics**

a. Sources/Parents:

b. Type = behavior quality

c. Design scope = crosscutting [local, crosscutting]

d. **Priority** = [essential, necessary, desirable]

f. Version =

g. Assignment:

h. Architecture-relevant = yes [yes, maybe, no]

STATES

a. Goal states are < @Unverified, Verified, Impl, Inactive>

# Quality goals are poorly understood

Required quality attributes (quality goals) are difficult to achieve and verify.

Most developers (and their managers) have little understanding of how to do either.

Developers understand testing, but not verification. Unfortunately, testing alone is inadequate for the verification of many quality goals. Quality verification may entail analysis, technical review, and measurement, as well as four modes of testing.

# Agile

Agile is a set of values and principles for software development. It is not a development methodology.

Agile methodologies embody Agile values and principles.

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

# Which Agile methodology?

There are 4 types of Agile methodologies

1. Named e.g. Scrum and XP
2. Pure-hybrid i.e. blend of 2 or more named methodologies
3. Mixed-hybrid i.e. blend of 1 or more named methodologies and non-Agile practices (e.g. defining quality goals up-front)
4. Relabeled i.e. calling whatever you are doing “Agile”

Each type has two forms:

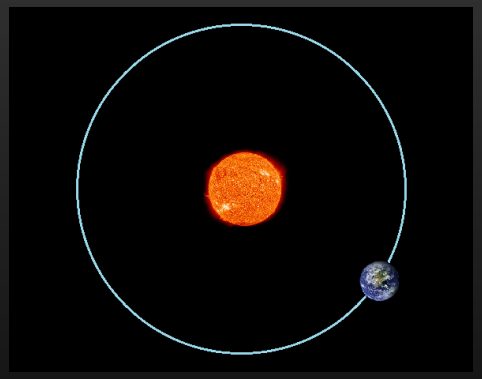
1. Written about i.e. Specified Agile (S-Agile)
2. Actually used i.e. Actual Agile (A-Agile)

Agile is a Quality Anti-Pattern means

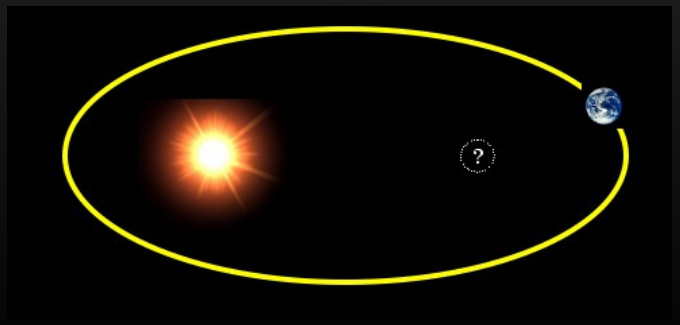
named or pure-hybrid S-Agile is a Quality Anti-Pattern

# Problem Definition -- Agile methodologies over-simplify


For every complex problem
there is an answer that is
clear, simple, and wrong.
H. L. Mencken



Ideas should be made
as simple as possible
-- but no simpler.
paraphrasing A. Einstein



# Agile's 4 problematic principles -- 1/2

## Welcome changing requirements, even late in development.

The impact of late changing requirements is NOT uniform. Since some quality goal supports are crosscutting e.g. exception handlers, if they are discovered or change late in development, achieving them may be infeasible or very expensive.

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

Sometimes, discussion alone is neither efficient nor effective. Larger scope topics such as safety achievement and verification strategies, must be recorded, analyzed, and then discussed. There are too many scattered elements to enable discussion alone to provide confidence in the assessment of either strategy.

# Agile's 4 problematic principles -- 2/2

## Working software is the primary measure of progress

The achievement of many quality goals is hard to assess except by verifying the entire application (e.g. when is safety working?)

Agile's "working software" often includes a lot of reckless short-term technical debt due to missing quality supports. Therefore, working software is not a measure, but an indicator, of progress. A thermometer measures temperature. The sweat on my brow indicates temperature.

## The best architectures, requirements, and designs emerge from self-organizing teams

Quality goals don't need to emerge. They can be selected at the beginning of a project from a rich attributes model. Emergence is great, when experience and understanding are lacking. It is ineffective and expensive, when the choices are known.

# Agile quality – what does it mean?

Scrum – the most popular methodology – provides no guidance on quality goals

Extreme Programming (XP) practices 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 – 2/3 not doing in 2010
- simple design
- refactoring

XP practices:

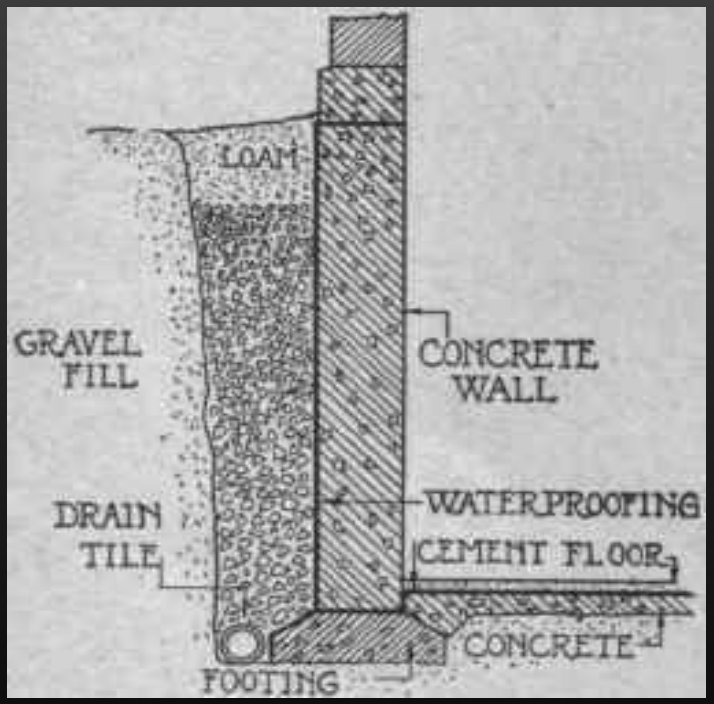
- focus on clean, understandable, and effective code thus supporting acceptable functionality, reliability, and understandability i.e. focus on 3 of over 50 attributes
- provide necessary, but insufficient support for most other attributes

# Agile's bottom-up functional bias

- Discourages specifications because code and tests are considered satisfactory
- Discourages up-front analysis and design for fear of waste, including gold-plating
- No specification or analysis of quality threats and (achievement and verification) strategies
- Focus on testing, rather than verification
- Emphasis on incremental design, which is very inefficient for crosscutting quality supports
- No mention of
  - risk management
  - resolving quality conflicts
  - designing crosscutting quality supports

# Functional bias is natural:  customers want functionality

# Developers (& their managers) neglect foundations




When a wall cracks (insufficient footings) in your home or your basement floods (insufficient drainage or waterproofing), you understand the need for a solid foundation.

When your system is hacked or crashes under high volume, you understand the need for a solid foundation i.e. set of quality supports.

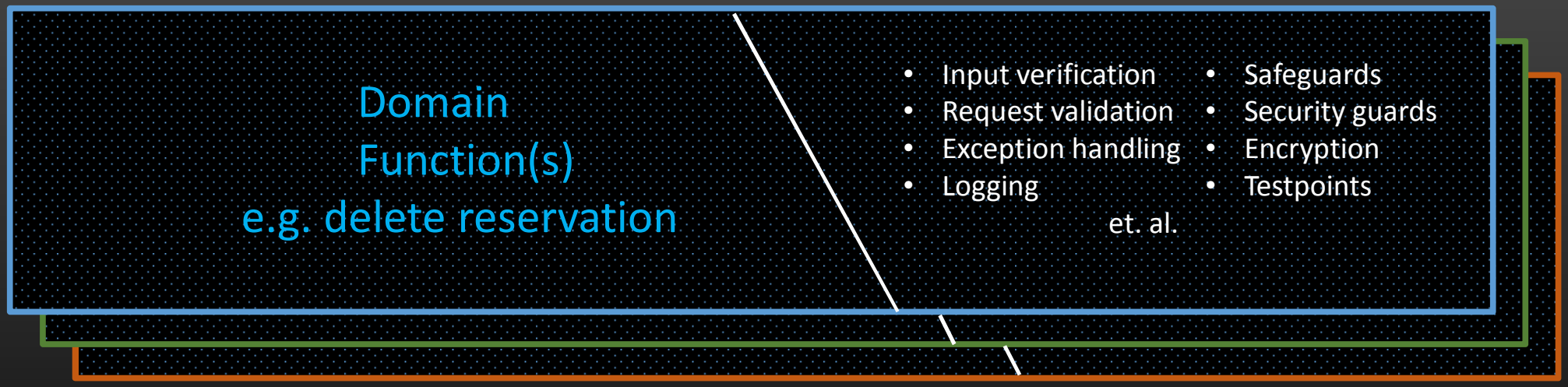
# Agile is terrible at achieving most quality goals

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

All Agile methodologies are terrible at achieving and verifying the other 50 quality goals, because:

1. Agile emphasizes functions and de-emphasizes most quality goals to the point of invisibility.
2. Agile emphasizes testing and, except for code and tests, de-emphasizes technical reviews, analysis, and measurement to the point of invisibility.
3. Nothing assures that all high-priority quality goals will emerge before product delivery.

# Agile* causes reckless short-term technical debt

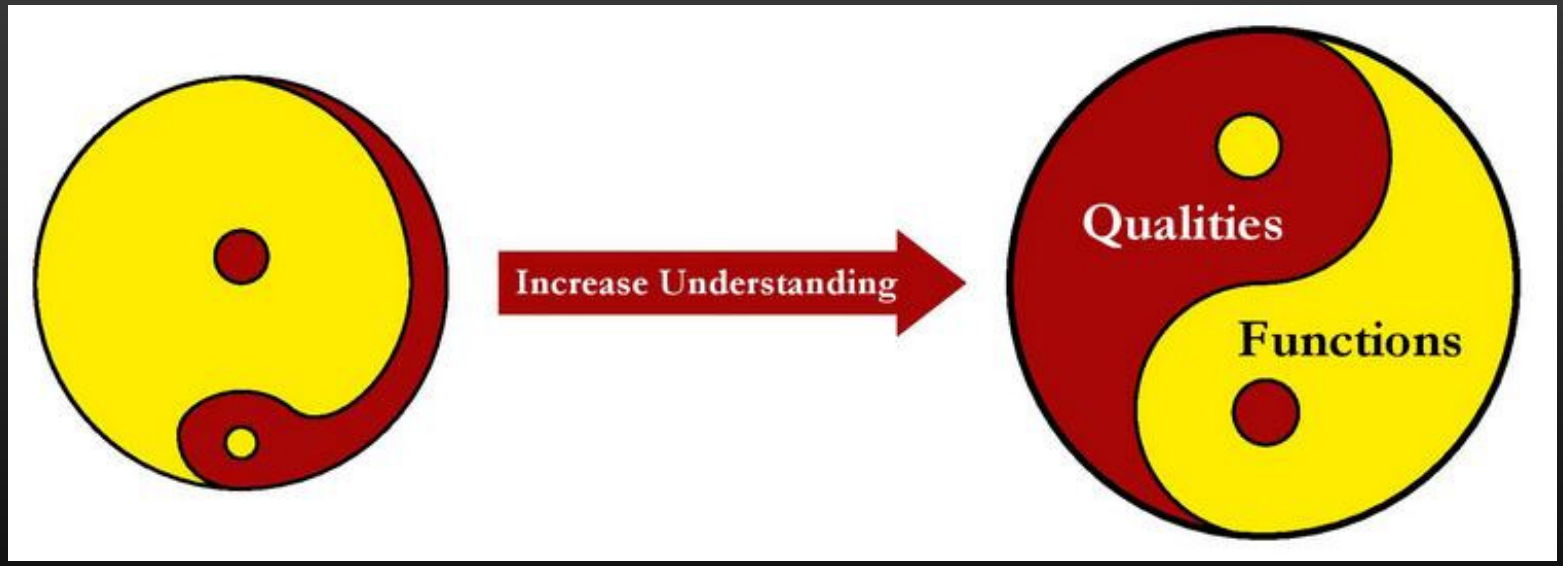



"Complete" functional components must contain quality support code

Late identification causes reckless short-term technical debt

Therefore, you should identify quality goals before functions

# Proposed Solution -- Quality before functionality



# Identifying quality goals first

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 rich quality attributes model.

Early identifications may need to be adjusted as understanding deepens, but there is no way to predict when you have 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.

# Quality-Aware Agile

Quality-Aware development is NOT a development methodology, but a 3-part supplement to whatever you are doing now or intend to do

It begins with a quality sprint in which you identify quality goals their levels, priorities, challenges, mitigations, supports, achievement strategies and verification strategies. You also acquire and verify a library of crosscutting support components e.g. exception handlers. Using a rich quality attributes model, it should take less than a week to draft a quality goals model.

In each iteration, you reassess and carry out the quality strategies.

Finally, you collect the quality lessons during a project retrospective and record them in the enterprise model of quality attributes and/or in the development standards.

# Voluntary ignorance

Voluntary ignorance is choosing NOT to acquire readily-available product requirements, implementation, and verification information in a timely manner.

Readily available means the information can be acquired “quickly” i.e. between one hour and one week.

Timely manner means that delay causes significant short-term technical debt.

Waiting for quality attribute requirements to emerge or reach the top of the backlog rather than using Quality-Aware Agile is an example of voluntary ignorance.

Replace voluntary ignorance with timely understanding

# Wrap up

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

We recommend Big Quality Requirements Up-Front (BQRUF) i.e. a mixed-hybrid strategy, because understanding of quality goals should not evolve much. Failure to practice BQRUF always results in significant short-term technical debt.

# An Offer

If you have an early stage project,
are interested in trying a "quality sprint", and
would like an expenses-only week of guidance,
let me know.

Questions or Comments

david@clearspecs.com