**SOME THOUGHTS ON REQUIREMENTS**

There are more things in heaven and earth, Horatio
than are dreamt of in your [requirements] philosophy.
Shakespeare

Let’s say you want cornflakes. You know the brand you like, the box size you want, and the price range you can afford. You could buy it yourself or ask someone else to make the purchase. If asking someone else, you need to communicate your wants. When communicating, you must provide details (requirements), such as what to do if the size is out of stock or what brand to substitute if they can’t find what you like. In this case, you know what you want, but you may need someone else to understand your wants.

Sometimes, you only know some of your wants. When the router in your home Wi-Fi network dies, you might go to an electronics store to learn about replacement options. You might ask why one router costs $200 and another $50. This helps you learn about various features and decide which are important to you.

Dealing with requirements is natural. Dealing with their comprehensive and precise specifications is not. This makes effective communication a challenge.

**Once upon a time**

Software requirements development (SRD) was simpler. Requirements shaped the goals of software development. Books and training taught that requirements should be comprehensively elicited, analyzed, and specified before coding. It seemed reasonable that goals should be defined before solutions.

This comprehensive specification approach held sway until the 1990s, when a paradigm shift declared that having comprehensive specs early doesn’t work. The focus shifted from creating an entire application to creating the next few increments. User stories, acceptance tests, and extensive customer conversations were believed to provide enough requirements information for the next increments. It was also believed that requirements and the application should be developed incrementally, partly because the developing system would deepen stakeholder understanding and thus clarify the requirements.

This change raised the questions:
- Exactly what is the role of requirements in software development and how should this role be performed?
- When, if ever, are written specs valuable?

From a historical perspective, it seems that requirements practice has shifted from one silver bullet (Waterfall) to another (Agile). As Fred Brooks [Ref. 0.1] told us long ago, in the increasingly complex reality of software development, there are no silver bullets.

He also told us there is a difference between essential and accidental complexity. **Essential complexity** is inherent in the problem to be solved and must be addressed by any effective solution. The challenge is to understand the essential complexity of requirements development and avoid the siren call of simplistic models and strategies.
Why is SRD so difficult?

If you had to choose between a root canal and developing requirements, which would you pick? It might be the dental work, because the pain interval is shorter and Novocain can help.

Why is SRD so difficult? It’s difficult—and always will be—because it is not consistent with our normal behavior. Its major challenges are not technical, but rather psychological and social.

The challenges of SRD include:
1. Unavailable or unfocused information sources.
2. Stakeholders unclear about their needs.
3. Stakeholders change their mind about what they want.
4. Disagreements about requirements and priorities.
5. Problems in making requirements precise.
6. Tacit stakeholder knowledge, which is hard to access.
7. Emergent understanding by stakeholders, which can’t be scheduled.
8. Poor understanding of quality attributes e.g., safety and security.
9. Concrete thinking by critical stakeholders, who can only identify problems in an implemented system, not in descriptions or models.
10. Misunderstandings, e.g., because of ambiguous communication.
11. Implicit requirements, which may escape detection during development.
12. Rapid changes in the application, domain or technology.

Most of these challenges relate to people and workings of the mind. This means that providing more time, buying new tools, or reorganizing a project team will rarely improve SRD.

While few projects face all of these challenges, many projects include some of them. Effective SRD begins with an early recognition of these challenges and the use of challenge-specific mitigation tactics.

When SRD is ineffective, a trial and error strategy often results. Even when this strategy works, the schedule and budget are usually compromised.

Backstory

I used to believe that specifying document content and task steps could improve software quality. While this approach has its place, decades of experience have taught me that complex problems need customized situation-specific strategies. For example, if developers are unfamiliar with an application domain, they must be guided to adequate understanding or replaced by developers who do understand. Sometimes, neither tactic will work within project constraints. Then, the project should have its scope reduced or be cancelled. A customized approach should be grounded in risk analysis and management; in other words, should entail figuring out what to fear and how to mitigate the risks.
Types of software requirements

Most requirements models only contain a few types. The following comprehensive model identifies many more. **Note the major role played by developers in requirements development.**

<table>
<thead>
<tr>
<th>Quality goals</th>
<th>Primary Sources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal qualities e.g., readability</td>
<td>Quality specialists</td>
</tr>
<tr>
<td>External qualities e.g., reliability</td>
<td>Customers &amp; Developers</td>
</tr>
<tr>
<td>Mixed qualities e.g., safety</td>
<td>Customers &amp; Developers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain functions</td>
<td></td>
</tr>
<tr>
<td>interactive</td>
<td></td>
</tr>
<tr>
<td>happy paths</td>
<td>Customers</td>
</tr>
<tr>
<td>unhappy paths</td>
<td>Developers</td>
</tr>
<tr>
<td>batch</td>
<td>Developers</td>
</tr>
<tr>
<td>autonomous</td>
<td>Developers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Quality support tactics</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>internal quality supports e.g., self-checkers</td>
<td>Developers</td>
</tr>
<tr>
<td>external quality supports e.g., internationalization functions</td>
<td>Developers</td>
</tr>
<tr>
<td>mixed quality supports e.g., safeguards</td>
<td>Developers</td>
</tr>
<tr>
<td>System functions e.g., backup</td>
<td>Developers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data</th>
<th>Customers &amp; Devs</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Constraints</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Technical</td>
<td></td>
</tr>
<tr>
<td>design e.g., no single point of failure, platforms, external interfaces and protocols</td>
<td>Developers</td>
</tr>
<tr>
<td>implementation e.g., coding standards, data restrictions</td>
<td>Quality specialists</td>
</tr>
<tr>
<td>verification e.g., test coverage</td>
<td>Verifiers</td>
</tr>
<tr>
<td>deployment e.g., mission configurations</td>
<td>Developers</td>
</tr>
<tr>
<td>Societal e.g., common practice</td>
<td>Quality specialists</td>
</tr>
<tr>
<td>Project e.g., deadlines</td>
<td>Project leads</td>
</tr>
</tbody>
</table>

| Supplier attributes | Quality specialists |
Precise definitions

Consider the following requirement for self-driving vehicle control:

“When people or people indicators or substantial objects are detected within the safe stopping distance, and safe swerving is not an option, then rapidly apply maximum pressure to the brakes.”

While developers may be familiar with driving, they aren’t familiar with the precise definitions of “defensive driving” terminology. Since these definitions don’t exist, they must be specified and learned.

Adequate domain understanding of defensive self-driving controls entails learning detailed definitions for the following named concepts:

- **entities**: people, people indicators (e.g., strollers), vehicles, substantial objects (e.g., jackknifed trucks)
- **entity relationships**: tailgating
- **values**: speed limit
- **conditions**: people indicators detected, roadway impassable
- **calculations**: safe stopping distance
- **hazards**: sink holes, kangaroos, fallen powerlines
- **procedures**: safe swerving
- **facts**: manual drivers move into significant open lanes
- **rules**: common US traffic laws in the current year

Stakeholders can learn domain concepts with the help of a domain knowledge base or glossary containing these types of definitions.
A useful quality model

Within the quality requirements development process, we find the **selection** of relevant quality attributes, the **specification** of relevant levels, and the **design** of achievement, verification, and monitoring strategies.

We can identify three dimensions of generic quality attribute models: lists, subgroupings, and details. All models have a list dimension. Some models are just a list, some are a list with details, some are a list with subgroupings, and some are a list with details and subgroupings. A list provides minimal help with selection, accurate subgroupings help to identify supporting qualities, and details may help with specification and strategy design.

In the figure below, a generic quality attribute model may be: not useful [], minimally useful [-], useful [+] or very useful [++] in dealing with quality attribute requirements.

<table>
<thead>
<tr>
<th>Models</th>
<th>Dimensionality</th>
<th>Usefulness</th>
<th>Select</th>
<th>Specify</th>
<th>Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBAP (IIBA)</td>
<td>List</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CSQA (QAI)</td>
<td>List</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Wikipedia</td>
<td>List</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ISO 25010</td>
<td>List, Subgroupings</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CPRE (IREB)</td>
<td>List, Subgroupings</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CSQE (ASQ)</td>
<td>List, Subgroupings</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Bass et. al., book</td>
<td>Software Architecture</td>
<td>List, Details</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Wiegers &amp; Beatty book, Software Requirements</td>
<td>List, Details, Subgroupings</td>
<td>+</td>
<td>++</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>LiteRM quality model</td>
<td>List, Details, Subgroupings</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td></td>
</tr>
</tbody>
</table>

The LiteRM quality model is freely available at [www.quality-aware.com/software-quality-KB.php](http://www.quality-aware.com/software-quality-KB.php) The model is described in **Understanding Requirements**, Chapter 3 – available on the same webpage.

Crosscutting requirements

**Crosscutting requirements** (such as most quality support functions e.g., safeguards) impact code in many components. **Local requirements** (such as the CRUD functions for domain objects e.g., reservations) only impact a few components. The scope of some requirements e.g., supplier attributes, is neither.

You should identify crosscutting requirements early because they place restrictions on many custom modules in a system. When a developer is creating a module to implement an application function e.g., make a reservation, they should also be aware of any associated crosscutting requirements e.g., complying with coding standards.
Domain functionality is just the beginning

<table>
<thead>
<tr>
<th>Domain Function(s)</th>
<th>Quality Support Tactics</th>
</tr>
</thead>
<tbody>
<tr>
<td>[e.g., create an order]</td>
<td>Self-checking, Exception handling, Logging, Safeguards, Security guards, Encryption, Test points, and other quality supports</td>
</tr>
</tbody>
</table>

Crosscutting achievement can’t be done in a single increment of development. Because the achievement of crosscutting requirements or goals is distributed, it is more difficult (expensive) to verify. Full verification needs the entire codebase. Failure to identify crosscutting requirements early causes reckless quality debt to pile up.

We distinguish three types of crosscuts: **universal**, **homogenous**, and **heterogeneous**.

Some requirements, such as understandability, are associated with universal crosscuts. These requirements restrict designs and most lines of code. A universal crosscut restricts all other crosscuts, including other universal crosscuts.

The implementation of a homogenous crosscut does the same thing wherever it appears. For example, a homogenous logging routine may appear at the exit of every module, but always collects and stores the same type of information.

The implementation of a heterogeneous crosscut does different things where it appears because its behavior is context-sensitive. The variety of security threats means that different security guards will be needed for each threat i.e., security entails heterogeneous crosscuts.

Failure to identify crosscutting requirements early or at all can be extremely expensive or economically infeasible to fix. Retrofitting universal crosscuts e.g., compliance with coding standards, is likely to be economically infeasible. From a risk perspective, heterogeneous crosscuts are dangerous, because some of their associated functions are likely to be complex or defective.

Incomplete and misunderstood quality attributes and goals are often major project risks. This risk should be managed with an early and continuous focus.

**The Why and What of QRUF**

Quality-Requirements-UpFront (QRUF) development reduces endgame surprises. This tactic mitigates the pervasive risk of significant quality debt.

Few developers have much experience managing the broad range of software quality attributes relevant to a software product. Their challenge is to:

1. **Select** the qualities relevant to the software under development along with levels and priorities. Selecting a quality entails selecting all the qualities that support it. For example, selecting safety entails selecting dependability, security, and many other qualities.
2. **Specify** goal levels and priorities for each selected quality.
3. **Identify** up to three candidate architectures based on the software’s core functionality as well as its contextual elements and constraints.
4. For each architectural candidate, **outline** achievement, monitoring, and verification strategies for the high-priority qualities **until a single architecture emerges**.

5. For the selected architecture, **design** an achievement, monitoring, and verification strategy for each quality including its quality-effort estimate. Achieving, monitoring, and verifying a quality entails achieving, monitoring, and verifying all its supporting qualities. Reference **stringent design and coding standards**, promoting software understandability, whose compliance will be verified. When appropriate, develop a **glossary of precise definitions** for domain concepts, verifiable quality measures, and terminology in the achievement and verification strategies. Use **assurance cases** to verifying the adequacy of the achievement and verification strategies for critical qualities and their supports.

6. **Carry out** the strategies to achieve, monitor, and verify each selected quality on each development increment.

QRUF development means doing as much of the first five tasks as possible at the beginning of a project so the last task can be carried out on each developmental increment. Unfortunately, at the beginning is just when most customers, managers, and developers are focused on application functionality. This is quite natural, since customers want software that does things. Customers may eventually want safety and dependability, but qualities are not the first things they think about. This makes QRUF development unnatural, but often necessary for risk and project cost reduction.

The LiteRM quality model or a tailored version helps with tasks 1, 2, 4, and 5 above. If a team is NOT using a tailored form of this model, then they are working too hard or not hard enough.

**Mixed strategies**

Uniform strategies are not effective in all software development situations. For example, consider the quantity of system-level requirements specification in different situations on different projects. Careful specification can manage understanding risk.

<table>
<thead>
<tr>
<th>Number of Specifications</th>
<th>Insourcing</th>
<th>Outsourcing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Few specs</td>
<td>Experienced/visionary developers Major reuse</td>
<td>Experienced developers</td>
</tr>
<tr>
<td>Some specs</td>
<td>Experienced/visionary users Developers &amp; users colearning</td>
<td>Experienced/visionary customers &amp; relevant examples</td>
</tr>
</tbody>
</table>
| Many specs               | Complex requirements Safety-critical applications | Experienced/visionary customers & no relevant examples
|                           |                                | Complex requirements Safety-critical applications |
Now consider different component-level situations on the same project.

<table>
<thead>
<tr>
<th>Component situations</th>
<th>Few specs</th>
<th>Reuse Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Some specs</td>
<td>Enhanced</td>
</tr>
<tr>
<td></td>
<td>Many specs</td>
<td>Complex Safety-critical</td>
</tr>
</tbody>
</table>

Prepare for customization by collecting a set of useful tactics with a balance sheet for each describing the tactic’s advantages and disadvantages. Then, design and follow a mixed strategy.

**Risk management**

Risk management is project management for adults.

Tom DeMarco & Tim Lister

Risk management is the least practiced of all project management disciplines across all industry sectors, and nowhere less applied than the IT industry.

Robert Charette

Since Charette found risk management to be rare, DeMarco and Lister might infer that project culture and project management practices are immature in many organizations. Some cultures reward firefighting, even when the fighters set the fires.

Much can go wrong during requirements development. Although poor requirements are just one type of project hazard, they can be a major source of problems.

**Requirements hazards**

Take time to review past projects, both yours and others. Development processes have characteristic defect and failure footprints, and you can learn a lot from them.

Defects and failures can be:

- **systematic** (caused by project practices and constraints)
- **natural** (caused by inexperience or human limitations)
- **accidental** (caused by human error)

If you plan to reuse a development process, consider its failure footprint and causes. This helps identify hazards that have caused problems in the past. In addition, use your experience and those of others to identify additional areas of concern.
The following generic lists of hazards to requirements do not highlight the hazards likely to appear on your project. The lists are meant to trigger recognition, based on your experience and intuition, of hazards you have seen.

**Domain**
- Unfamiliar
- Complex

**Application**
- Unfamiliar
- Complex
- Safety or mission critical

**Stakeholders**
- Incorrect assumptions
- Insufficient or incorrect knowledge
- Insufficient experience or understanding
- Disruptive behavior
- Conflict
- Critical stakeholders overlooked, unavailable, or under-involved

**Process**
- Scope too broad or too narrow
- Scope creep
- Inadequate discovery
- Incorrect derivation
- Inadequate focus on quality goals
- Inadequate focus on supplier attributes
- Inadequate focus on constraints
- Confusing responsibilities
- Context-insensitive tasks
- Ineffective communication or negotiation
- Too little or too much specification
- Inadequate validation
- Ineffective information or change management

**Environment**
- Unsafe
- Distributed

**Resources**
- Inadequate time for discovery, analysis, communication, or validation
- Inadequate tool support

**Changes**
- Unstable scope
- Unstable requirements
- Crosscutting changes discovered late
**Requirements mitigation tactics**

After identifying requirements hazards needing attention, identify tactics that can mitigate them. If you think your controls might be inadequate, consider adding a few of the tactics from the list below or developing others.

**A. Monitor feasibility of requirements and project**

Projects become “famous failures” by promising much, consuming many resources, and delivering little or nothing. This category focuses on avoiding this infamy.

- Reconfirm reality and satisfiability of stated needs
- Reconfirm project feasibility or reduce scope
- Assess stakeholder understanding
- Triage and prioritize requirements
- Manage customer expectations
- Identify and resolve conflicts early
- Commit incrementally

**B. Prevent requirements defects**

This category focuses on deepening stakeholder understanding and developing an effective RD process to reduce the likelihood of mistakes.

- Maintain a climate of personal safety, respect, and cooperation
- Manage stakeholder understanding
- Locate external experts with a deep understanding of opportunities or solutions
- Study solutions to similar problems
- Have developers maintain similar applications
- Immerse stakeholders, i.e., customers in system development and developers in customer activities
- Model usage
- Create an application domain model
- Select quality goals from a useful quality model to reduce requirements effort
- For intentionally vague requirements, identify alternative solutions and their costs
- Prototype unfamiliar functions and interfaces
- Promote collaborative research and cooperative learning
- Customize a “just enough” information strategy
- Study and classify past requirements defects
C. Minimize communication problems

Effective project communication is a core challenge in system development. Natural language is a major problem, because it is familiar and ambiguous. This category describes alternatives and supplements to text blocks.

- Isolate “well-known” details
- Use a domain glossary with verified entries to reduce vagueness and identify and minimize terminology overlap
- Replace “magic numbers” i.e., constants, with named constants or formulas
- Identify user types and create user personas
- Clarify with examples and measures
- Express information with spec patterns—avoid text blocks
- Capture context

D. Detect and correct requirements defects

Validating requirements can be a challenge. This category makes some suggestions. Also see chapter 5.

- Organize functional requirements and constraints by objects, groups, and then functions to assist semantic overlap detection
- Identify vague, incorrect, missing, and unnecessary information
- Analyze verification strategies and test designs
- Assess developer understanding

E. Monitor requirements and their results

Requirements and their associated information (e.g., defects) must be monitored to detect unexpected risks. This category describes tactics for this monitoring.

- Monitor requirements instability and growth as well as their major causes and project impacts
- Monitor accuracy of assumptions, expectations, and priorities
- Monitor stakeholder participation
- Require frequent demos of developer understanding
- Trace requirements derivations and links to resulting work products
- Analyze and classify requirements defects and root causes
- Use external, mid-project audit of hazards for a fresh perspective
- Include requirements risk reassessments in your project status reviews