Chapter 6. Checking Requirements

If it’s worth specing, it’s worth checking.

Since some requirements-related misunderstandings can be expensive, and “to err is human”, it makes economic sense to check requirements.

Checking requirements entails skeptical technical reviews, analysis, measurement, and design of development and verification strategies. When requirements are used to create user or operator manuals and requirements-based tests, this usage may uncover requirements defects.

Defective quality goals or technical constraints can cripple an application and be much more expensive to correct than defective functional requirements. This suggests that quality goals and constraints should be validated early and often.

Since stakeholders are more likely to overlook rare or unusual conditions, these should receive increased attention. For example, in the specs for the International Space Station, missing and incomplete information defects represented 54% of the requirements problems and incorrect information accounted for another 24%. Although the Space Station is unique, this application illustrates how looking for missing, incomplete, and incorrect requirements is likely to be a cost-effective strategy.

6.1 Requirements information fragments

We begin by summarizing the requirements information fragments (RIFs), since these are work-products that should be checked.

Non-functional specs

- Quality goal specs
- Constraint specs
- Supplier attribute specs

Functional specs

- Scope statements
- User stories
- Logic tables
- State transition tables
- Action contracts
- Intent-Explicit Behavior specs
- User and operator manuals
- Free-form natural language
- Controlled natural language
- Requirements modeling languages
- Readable Tests
- Prototypes
- Application code

Domain specs

- Object specs
- Class diagrams
- Facts about values and conditions
- Entity-relationship diagrams
- Domain glossary
- Harmful outcomes
- Hazard analysis results
- Fault Tree analysis results

Background and context specs

- Rationale and assumptions
- Operating environment specs
    - Context diagrams
    - Data flow context diagrams
    - Interface specs
- Usage and misusage patterns

6.2 Attributes of good RIF specs

Each good RIF should have the following characteristics:

- **Discrete** – distinct from other RIFs
- **Understandable** – easily understood by stakeholders with the appropriate background and experience
- **Unambiguous** – has a single interpretation
- **Valid** – accurately reflects context or stakeholder, application, or project needs or wants
- **Necessary** – is within the project scope
- **Complete** – contains all task-adequate information at a sufficient level of precision
- **Concise** – contains only necessary information
- **Conformant** – format complies with regulations, standards, or guidelines
- **Feasible** – achievable within project constraints
- **Verifiable** – achievement is assessable within project constraints

In addition, a set of RIF specs should have the following qualities:

- **Task-adequate** – contains all RIFs that need to be specified
- **Concise** – contains only necessary RIFs
- **Consistent with each other** – in content and format
- **Prioritized** – organized into groups based on importance
- **Feasible** – achievable within project constraints including budget and schedule

6.3 Types of RIF defects

See the types of requirements defects in section 4.1.

6.4 Checking tactics

We first consider general tactics that apply to all RIFs. Then, we consider tactics specific to functional requirements and quality goals.

6.4.1 General tactics

A. **Skeptical technical reviews** – This entails using checklists for content and structural defects as well as using paraphrasing and "term defining" in group sessions [Ref. 7.1]. Reviewers assume each work-product is defective and search for the defects.

- Include knowledgeable and experienced users and customers to identify inaccurate, incomplete, and unnecessary information
- Include experienced developers to identify infeasible and unverifiable objectives
- Acquire general and spec-specific checklists to aid reviewers
- Walk through critical scenarios
- Review relationships
    - Identify and check implications, e.g., "system must warn of high temperature" implies system must acquire temperature data in a timely fashion
    - Identify dependencies between requirements
- Determine the purpose of each RIF to reveal those that are unnecessary
- Identify unnecessary constraints and designs
- Review clarity
    - Locate vague and undefined terminology
    - Locate ambiguous expressions [Ref. 7.2]
    - Simplify wording and grammar
    - Breakup some compound expressions
    - Check for inconsistent forms of expression
- Find "To Be Dones" (TBDs) and invalid references
- Confirm "Not Applicables" (NAs)
- Review hazard analysis and fault tree analysis of mission-critical functionality and quality goals
- Question missing information that is expected and unexpected information that is present
- Check for noncompliance with content and structural standards and guidelines

B. **Analysis**

- Use spelling, grammar, and style checkers
- Use vague and ambiguous language checkers [Ref. 7.3]
- Check constraints on controlled natural languages and requirements modeling languages
- Check the understandability of each RIF
- Check the validity of each formatted RIF

C. M**easurements**

- Measure readability
- Sample defect density [Ref. 7.4]

Templates, best-practice examples, and guidelines make assessing completeness, need, and clarity less subjective. They may enable automated analysis of these characteristics.

6.4.2 Functional requirements tactics

A. **Skeptical technical reviews**

- Formally review for appropriateness, feasibility, sufficiency and scope, e.g., too broad or narrow
- Identify user roles that are unserved or under-served

B. **Analysis**

- Identify missing and incomplete functions with limited use of discovery tactics [Ref. 7.5, 7.6]
    - For breakdown structures:
        - Top goals should be assessed for sufficiency and necessity
        - Each breakdown should be analyzed to determine whether each set of child goals will achieve its parent goal and whether each child goal is necessary
    - Use scenarios and cases to:
        - Identify missing functions
        - Model rare use, extreme use, and misuse
    - Consider the needs of secondary users to identify missing functions
    - Consider periodic housekeeping functions on data, e.g., the archiving of “old” transactions

    - Group conditioned rules into tables and look for missing conditions or overlap
    - Group specs to uncover inconsistencies, e.g., when x, do a and when x, don’t do a; omissions, e.g., creation, but no deletion, and redundancies
    - Consider prototyping questionable functionality
- Analyze use of “and’s” and “or’s” for accuracy
- Analyze use of quantifiers, i.e., “all”, “every”, “each”, and “never” for accuracy
- Analyze clarity of negative specs
- Analyze formal specs using theorem proving, model checking, or test-case generation [Ref. 7.7, 7.8]

C. **Measurements**

- Calculate function points to assess clarity and completeness of requirements and to determine cost-benefit

D. **Test design**

- Design or automatically create requirements-based tests

Designing tests can be used to check functional specs, prototypes, or production code. When some functional tests are missing, but functional requirements are specified, design the missing tests from the specs. When some functional tests are missing, but a prototype or production code is understandable, reverse engineer some of the missing tests. Reverse engineering is a tactic that may reveal problems not easily seen in the code. If both understandable code and functional specs are available, but some functional tests are missing, check both by designing some of the missing tests.

6.4.3 Quality goal tactics

- Identify missing quality goals using a quality attributes model
- Identify conflicting quality goals
- Identify appropriateness of quality levels and priorities
- Assess effectiveness of development and verification strategies

Bottom line

Checking is included in the cost of good requirements information. If the requirements information is important, the cost of checking is much less than the cost of poor requirements.

Supplementary readings (References)

Papers are available at understandingrequirements.com/Chapter-5-references.php.

6.1 Shull, Forrest et al. (2000) “How Perspective-Based Reading Can Improve Requirements Inspections” *IEEE Computer*

6.2 Bender, Richard (2003) “The Ambiguity Review Process”

6.3 Femmer, Henning et al. (2014) “Rapid Requirements Checks with Requirements Smells**”** **ICSE ‘14**

6.4 Gilb, Tom (2005) “Agile Specification Quality Control” *Cutter IT Journal*

6.5 Friedrich, Wernher and van der Poll, John (2007) “Towards a Methodology to Elicit Tacit Domain Knowledge from Users” *Interdisciplinary Journal of Information, Knowledge, and Management*

6.6 Hope, Paco et al. (2004) “Misuse and Abuse Cases” *IEEE Security & Privacy*

6.7 D’Silva, Vijay et al. (2008) “A Survey of Automated Techniques for Formal Software Verification” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*

6.8 Bowen, Jonathan P. and Hinchey, Michael G. (2006) “Ten Commandments of Formal Methods … Ten Years Later” *IEEE Computer*