

Introduction to Software Requirements Engineering

Part 1: Domain Knowledge

Björn Regnell

Updated: 2024-03-18

License: CC-BY-SA

<https://github.com/bjornregnell/reqeng-book>

Part 1: Domain Knowledge

Outline

Introduction

Specification

Context

Elicitation

Prioritization

What is Requirements Engineering (RE)?

- RE is focused on the
 - **features** of software systems
 - **system context**, including users and connected systems
 - **development context**, including stakeholders' intentions
- The RE process involves
 - knowledge-building
 - consensus-building
 - decision-making
 - innovation
 - communication



research
agree
choose
generate ideas
be pedagogical

What is a requirement?

- A simple definition:
 - Something **needed** or **wanted**.
 - A documented **representation** of something needed or wanted.
- Are we representing what is **actually** needed or wanted?
- '*Requirement*' can in practice mean many different things: must, option, idea, innovation, intent, rationale, function, quality, design, feature, decision, constraint, ...
- The most **general** meaning:
any kind of **information entity** used in RE



Core Activities of RE

- The 4 core activities of RE are:

- **Elicitation** learning
- **Specification** representing
- **Validation** checking
- **Selection** deciding

- In practice, these activities are often

- **Interdependent** output of one is input to others
- **Concurrent** one activity triggers others
- **Continuous** throughout the product's life as it evolves

What is good RE?

- Feasible and helpful foundation for software development
- Cost-effective process with high artifact quality
- Happy stakeholders
- Good system
 - commercially successful
 - beneficial to its users
 - ethical, helpful to society
- When are we ready? What is good enough?

RE in the Development Process

- RE interprets stakeholders intentions into validated req specs
- RE provides input to, and learns from down-stream activities
 - System Design
 - Quality reqs determine architectural decisions
 - System Implementation
 - Functional reqs (data and logic) are realized in code
 - System Verification
 - The req spec define correct output in test cases
 - System Operation
 - User feedback is input to requirements evolution
- As requirements evolve you must manage impact of changes
- Traceability:
 - Links among artifacts to support change management
 - Forwards: from requirements to down-stream activities
 - Backwards: from requirements to stakeholders

Requirements as Solution Constraints

- U: the **universe** of all possible software systems
- S: the **solution space**, a subset of U including all systems that **fulfill the spec**
- S contains both "**good**" and "**bad**" systems
- The **general purpose** of RE:
 - to **constrain the solution space** so that software development is likely to produce a **good enough** solution
- The req spec should be a good enough definition of what we mean with a "good enough solution"
- RE is the **foundation for software quality**.

Common Acronyms

■ RE	requirements engineering
■ SE	software engineering
■ SW	software
■ HW	hardware
■ FR	functional requirements
■ QR	quality requirements
■ SRS	software (or system) requirements specification
■ req	requirement
■ spec	specification
■ constr	constraint
■ sys	system
■ dev	development
■ ops	operations
■ org	organisation

What is a Requirements Specification?

- A simplistic definition:
 - *"A document that describes what the system should do"*
 - what is what and what is how?
 - how much about the context is needed?
 - not always a document; database, issue tracker, prototype, ...
- A collection of requirements models + Help for the reader
- Expressed using a combination of suitable media, such as:
 - text
 - diagrams
 - prototypes
 - test cases
 - videos
 - ...
- Similar to a shopping list:
 - You don't always get what you want.
 - You often want things that you don't need.

Different kinds of requirements

- (Parts of) Requirements are often labeled as:
 - **Functional Requirements (FR)**, including:
 - Requirements on **Data**
 - Requirements on **Logic**
 - **Quality Requirements (QR)**
 - Accuracy, Capacity, Performance, Reliability, Usability, Safety, Security, ...
- In practice FR and QR are often **combined** and **related**:
 - Functions have quality:
 - a function can be unreliable and unsafe due to bugs
 - Logic and data is related:
 - functions have input, state, output
 - Quality is supported by functions:
 - a login function supports system security

Requirements at different levels

- Level of **design abstraction**: from 'why' to 'how'
- Level of **detail**: amount and richness of information
- Level of **aggregation**: grouping, hierarchical decomposition
- Level of **formality**: from unstructured to mathematical

Abstraction on the Goal-Design-scale

why → *what* → *how*:

- **Goal-level:** *why*?
 - focus on intentions of stakeholders and users
- **Domain-level:** *what* do users do with the system?
 - focus on usage context of a feature, normal and exceptional usage, domain events
- **Product-level:** *what* does the system do?
 - focus on system behavior, input-logic-state-output, normal and exceptional input/output, product events
- **Design-level:** *how*?
 - up-front design choices, implementation details
 - really required/justified? often better as example only, not req

Which level is best? It depends. They are often combined.

- Too much 'how' may over-constrain the solution space giving too little freedom for developers to find the best solution.
- Without 'why' the risk of an unsuccessful solution is high.

Levels of Formality

From unstructured to mathematical:

- Very informal: free-form representation, no explicit rules
- Very formal: syntax, semantics, inference, meta-language
- Pragmatic middle-ground: restricted natural language + diagrams with explanations
- Pro: Formality enables automatic checks, concise models, ...
- Con: Formalization requires effort, knowledge, skills, ...

Level of formality? – a difficult tradeoff

- Formality in various aspects to a varying degree:
 - Very informal: free-form representation, no explicit rules
 - examples: slide presentation, textual narrative
 - Very formal: formal syntax, operational semantics, inference
 - examples: state machine, regular expression, predicate calculus
- Advantages of formalization:
 - Reduced ambiguity
 - More concise models
 - Enables tooling: automatic checks, proof of soundness, ...
- Disadvantages of formalization:
 - Harder to understand
 - Requires effort, specialized knowledge and skills
 - Limited in scope and expressive power
 - Some stakeholders cannot contribute in validation

Explicit or implicit requirements?

■ **Explicit** requirement:

- has a unique id, such as a mnemonic (short name) or number
- often has status, priority, or similar
- often has an explicit "shall"-statement
- often has links to other related spec parts with id

■ **Implicit** requirement:

- part of spec but no id, no status, no "shall"
- is text/diagram a requirement or just help for the reader?

■ Advice:

- Make most important requirements explicit.
- Link diagrams to explanatory text with explicit requirements.

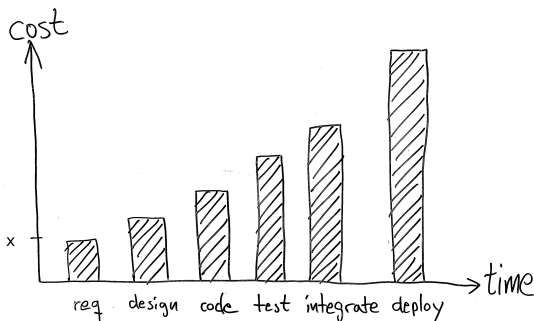
What is a good enough requirements specification?

Example of **quality factors**:

can only be achievable to some degree; can be conflicting

- **Correctness**: represents the stakeholders' intentions
- **Unambiguity**: stakeholders have similar interpretation
- **Completeness**: most of important relevant aspects included
- **Consistency**: no contradictions among requirements
- **Conciseness**: suitable level of abstraction and detail
- **Comprehensibility**: understood by stakeholders
- **Verifiability**: possible to check fulfillment
- **Feasibility**: possible to implement, value to justifiable cost
- **Traceability**: reqs can be referred to, can find origin of reqs
- **Modifiability**: easy to change, good structure
- **Ranked**: includes assessment of importance and stability

Cost of RE defects



- The cost of req defects increase with time.
- A req defect that costs x to fix in req validation may cost $100x$ or even $1000x$ in production.
- Why may cost of req defects increase exponentially?
 - Number of dependent artifacts multiply over time
 - If the foundation changes, many things need to be updated

How to best do RE is highly context-dependent

Aspects of the RE context to consider:

- **Stakeholder configuration:** relation customers – supplier
 - Examples of customers (users) and suppliers (developers): *public authority, private consumer, individual contributor, company (system integrator, subcontractor), community, company, company-internal department, ...*
- **Business model:** risk-sharing, profit-sharing:
 - internal budget, license fee, subscription, freemium, ad-based, donations, open-source community, non-profit, ...
- **Delivery model:** one-off, eventually updated, continuous integration and delivery
- Questions regarding customer–supplier relation:
 - Who has the knowlogshould
 - Who has the poeshould
 - Who gets the biggest value/profit? short- vs longoeshould
 - Who takes the bogshouldt risk?

Type of product

- Level of customization
 - generic
 - customer specific
- Hardware integration:
 - HW+SW
 - Pure SW
- Network integration
 - off-grid
 - connected
 - distributed
 - concurrent massive multi-user online communication, ...

Examples of common RE Contexts:

- Public tender: a public authority invites suppliers to bid
- B2B: both customer and supplier are companies
- B2C: the supplier provides SW to a consumer market
- In-house: one org develops system for internal use
- Open-source library: organisations share SW investments
- Embedded system
- Webb app: backend-frontend
- High-assurance systems: security and safety is critical

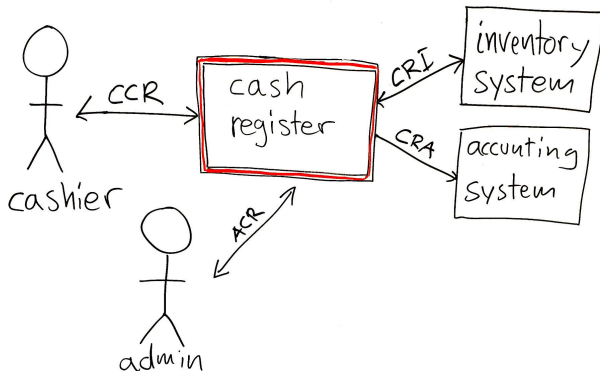
Scale of RE

- The RE effort increases exponentially with size!
 - Number of requirements: N
 - Number of pairwise relations: $R = N(N - 1)/2 \approx N^2$
- Orders of magnitude:
 - **Small-scale RE:** $N \approx 10^1, R \approx 10^2 = 100$
 - requires small effort, all pairwise relations can be considered
 - **Medium-scale RE:** $N \approx 10^2, R \approx 10^4 = 10\ 000$
 - feasible but requires large effort, consider subset of relations
 - **Large-scale RE:** $N \approx 10^3, R \approx 10^6 = 1\ 000\ 000$
 - unfeasible unless requirements are bundled into groups with high cohesion within groups and low coupling across groups
 - **Very large-scale RE:** $N \approx 10^4, R \approx 10^8 = 100\ 000\ 000$
 - unfeasible even if requirements are bundled into groups as the groups become either too many or too large
 - feasible only if the system can be split into subsystems with independent RE

Context Diagram

- A diagram describing the environment of the product
- The named product in the center as a **closed** box
 - no internal structure is shown – the focus is on context
 - open box with system parts inside is an *architecture* diagram
- Entities interacting with the product are connected by arrowed lines to show data flow direction
 - User roles (actors), shown as straw man icons
 - Other connected systems, shown as named closed boxes
- **Inner domain:** *direct* interaction with product
- **Outer domain:** *indirect* interaction with product
 - often *not* included in the context diagram
- Accompanying explaining text, including explicit requirements: "the system shall have interface X"

Context Diagram Example



* Interface CRA:

* Spec: The system shall ... data entities ...

What is requirements elicitation?

- Engaging with stakeholders
- Building domain knowledge
- Discovering and inventing requirements
- Exploring contextual usage
- Starting-points:
 - Stakeholder Analysis
 - Context Diagram
 - Product Scoping

Why is elicitation so hard?

Elicitation challenges: Stakeholders often...

- cannot abstract
 - difficult to explain what they do and why
 - difficult to express what they (really) need
 - ask for specific solutions
- lack imagination
 - of new ways of working
 - of consequences of new solutions
- complicate the picture
 - have conflicting demands
 - actively resist change
 - have luxury demands, "gold plating"
 - have new demands once others are met

Elicitation Methods

- Overview of elicitation methods:
 - Surveys
 - Interviews
 - Case-studies, examples: demos, usability tests
 - Creativity methods, example: brainstorming, focus groups
 - Operation Data Analysis: example: telemetry
 - Business Intelligence: observing competitors
- Elicitation methods support specification, validation and selection, example: focus groups support selection, usability testing support validation

Surveys

- Good for asking many persons to get an overview of distribution of views
- **TODO!!!** topics to consider
 - Population definition and sampling
 - Response rate
 - Closed and open questions
 - Lickert Scale
 - statistics, correlation, etc.

Interviews

- Unstructured interviews: open questions, open topics
- Structured interviews: closed questions, focused topics
- Semi-structured: combine both

Case Studies with Stakeholders

- Demonstrations by stakeholders
 - task enactment in a specific usage context
- Observation of stakeholders
 - sometimes it is easier to show than tell
- Prototyping (has its own chapter)
- Usability testing (has its own chapter)
- Pilot product deployment
 - limited but real usage of system in production
 - sometimes deployment is higher risk than development

Operation Data Analysis

Observe system in production before subsequent evolution

- Usage statistics, telemetry
- Online user experiments, A/B-testing
- Feedback from marketing
- Feedback from support
- Engage with user communities
- Mining social media

Creativity Methods

Group activities that support innovation.

- Purposes:

- trigger change and give competitive advantage
- facilitate stakeholders in idea generation and assessment
- get feedback on novelty and market opportunities

- Example methods:

- Brainstorming: free-form idea generation without assessment
- Focus-groups: structured brainstorming with assessment
- Creativity Workshops: explore, combine, transform

Creativity Workshops

Workshops based on applied creativity theory including:

- Exploratory phase: opening up the space of ideas
- Combinatorial phase: combining ideas to generate new ones
- Transformational phase:
 - change problem space so something that is impossible now becomes possible
- Analogical reasoning:
 - transfer knowledge from analogical domain
- Storyboarding:
 - integrate ideas related to selected use cases

Requirements Selection

- Requirements Selection provide input to downstream activities, answering the question:
 - What features are currently in and out of scope?
- Requirements selection includes:
 - **Prioritization**
 - ranking of requirements based on aspects such as benefit, cost, risk
 - **Product Scoping** (own chapter)
 - Defining the scope and theme of each release
 - Release Planning: deciding the feature set included in each release, while taking into account resource constraints and priorities

Why Prioritize?

- To focus on the most important issues
- To find high and low priority requirements
- To implement requirements in a good order
- To save time and money

Prioritization steps

- Select prioritization aspects (e.g. benefit, cost, risk)
- Select prioritization objects (e.g., features)
 - Try to define features at a high-enough level that can be selected or de-selected independently (if possible)
- Structure and groups objects
- Do the actual prioritization
 - Decide priorities for each aspect, for each object
- Visualize, discuss, iterate...

Why is prioritization hard?

Prioritization challenges:

- Finding a good abstraction level
- Combinatorial explosion
- Inter-dependencies
- Not easy to predict the future
- Power and politics

Prioritization Aspects

Examples of prioritization aspects:

- Importance (e.g. financial benefit, urgency, strategic value, market share...)
- Penalty (e.g. bad-will if requirement not included)
- Cost (e.g., staff effort)
- Time (e.g., lead time)
- Risk (e.g., technical risk, business risk)
- Volatility (e.g. scope instability, probability of change)
- Other things to consider:
 - competitors, brand fitness, competence, release theme
- Combine and optimize aspects, e.g.:
 - cost vs. benefit, cost vs. risk, importance vs. volatility
 - maximizing benefit while minimizing cost

When to prioritize?

- Before spending large RE effort on a specific feature
- At decision points, e.g.,
 - Start of feature design
 - Start of feature implementation
 - Release Planning
- When big changes occur
- Regularly with *lagom* intervals

Who should prioritize?

Find the right competence for the right aspect

- **Developers** know about e.g.,
 - development effort and engineering risk
- **Support** organization knows about e.g.,
 - customer value if included and cost penalty if excluded
- **Marketing** organization knows e.g.,
 - competitors' products, market opportunities, cost of sales
- etc...

Prioritization Scales

- For each aspect you need to decide on a metric
- A metric is expressed/estimated using a value on a **scale**
- Different types of scales with different power:
 - categorical scale $\{A, B, C\}$
 - example: {must, ambiguous, volatile}
 - ordinal scale $A > B$
 - examples: higher value, more expensive
 - ratio scale $A = k \cdot B$
 - examples: amount of money, hours, percentage

Prioritization Methods

Different methods, can be combined

- Grouping, categorical scale
 - example: use post-it notes on a white-board to group interdependent features
- Top- N , e.g. $N = 5$, categorical combined with ordinal
 - example: select 5 most beneficial features from the viewpoint of a specific stakeholder
- Grading on an ordinal scale
 - examples: grading 1...5, high-medium-low
- Ordering by pair-wise comparison (sorting, ordinal scale)
 - use insertion sort to arrange in order of highest to lowest risk
- 100-dollar-test, ratio-scale
 - distribute fictitious money to reflect prioritization aspect

Prioritization Methods Combined

Example of how prioritization techniques can be combined:

- Start with a high-level grouping of features that are highly interdependent to reduce the number of prioritization objects
- Sort all features of small groups in benefit order
- Use Top-5 for groups with large number of features
- For selected groups that are most important for the coming release: do a ratio-scale prioritization with the 100-dollar-test

Prioritization as Constraint Solving

- **TODO!!!** Part of lab
- **TODO!!!** discuss circular inconsistency in pair-wise comparison