Requirements Engineering

Bernd Fischer
bfischer@cs.sun.ac.za

Requirements engineering is difficult

“The hardest single part of building a software system is deciding precisely what to build. No other part [...] is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part [...] so cripples the resulting system if done wrong. No other part is more difficult to rectify later.”

F. P. Brooks

Topics

• requirements
• requirements errors
• requirements specification

Requirements engineering

Main concern: what should the system do?

Main result: software requirements specification

== description of desired system functionality

Requirements engineering is difficult

Experiment: formal inspection of requirements (10 pages) for centralized railroad traffic controller

• written by experienced project leader in domain
  – author believed that teams would find only few errors
  • inspected by 10 teams of 4 software engineers
  • 92 errors, some very serious, were found!
  • each team found only 35.5 errors on average!
    – i.e., it missed 56.5 to be found downstream
  • many errors were found by only one team!
  • errors of greatest severity found by fewest teams!

More information:

J Martin, WThoF Fold Inspection: A Requirements Analyse Technique. CACM 33(2) 225-232 (1990)
**Requirements engineering is difficult**

- 80% of interface faults and 20% of implementation faults due to requirements [Perry & Stieg 1993]
- 85% of defects due to requirements
  - 49% due to incorrect assumptions
  - 29% due to omitted requirements
  - 13% due to inconsistent requirements [Young 2001]
- 1.9 faults per page of specifications,
  0.9 faults per page of design,
  0.3 faults per page of code! [JPL]
- 54% of all errors found after coding and unit testing
  - 83% due to requirements and design
  - 17% due to coding stage [Boehm]

**Requirements errors are common and stubborn.**

So: requirements errors...

- ... cause most defects
- ... are relatively more common than code errors
- ... are found out late

**Requirements errors are common, stubborn, and expensive.**

Requirements errors are more expensive to fix later:

- fixes in requirements can be done in text
- other fixes require
  - localization
  - change of
    - design
    - code
    - documentation
  - test of change itself
  - system regression test
- worst case: system re-design and re-development [Stecklein 2004]

**Requirements errors are common, stubborn, and expensive.**

Requirements errors are more expensive to fix later:

- fixes in requirements can be done in text
- other fixes require
  - localization
  - change of
    - design
    - code
    - documentation
  - test of change itself
  - system regression test
- worst case: system re-design and re-development

---

**Requirements**

- **Requirement**: a thing that is needed or wanted.  
  Oxford Dictionary
- **Requirement**: a singular documented physical and functional need that a particular design, product or process must be able to perform.  
  Wikipedia
- **Requirement**: a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.  
  IEEE Standard Glossary of Software Engineering Terminology
**Requirement**

A requirement is a description of one service or constraint of the system that the user desires.

Requirements are...
- ... aimed at different audiences: user requirements vs system requirements
- ... aimed at different aspects of the system: functional vs non-functional vs domain
- ... described with different formalisms: natural language vs graphical vs formal methods

**User requirements**

- written primarily for system users who do not necessarily have detailed technical knowledge
  - managers & users (customer) – to define system
  - managers (contractor) – to plan system development
  - developers (contractor) – to specify system reqs.
- often in natural language plus diagrams
- often complemented with rationale to help developers understand
  - the application domain
  - why the requirement is stated in its current form
  - possible effects of requirements changes
- also called requirement definition

**User requirements - example**

1. The library information system shall allow users to search books and audio-visual materials through the catalogue page.
   - needs refinement
   - which catalogue?

**System requirements**

- written primarily for system developers as starting point for design
- structured document setting out detailed, technical descriptions of the system services
  - adds detail to the user requirements
  - should be consistent with it
- sometimes also used as a contract between client and contractor
  - tension between precision and detail (for design) and openness (for contracting)
- also called requirement specification

**Example**

**User requirement**

1. The software shall provide a means of representing and accessing external files created by other tools.

**System requirements**

1.1 The user should be provided with facilities to define the type of external files.
1.2 Each external file type may have an associated tool which may be applied to the file.
1.3 Each external file type may be represented as a specific icon on the user’s display.
1.4 Facilities should be provided for the icon representing an external file type of be defined by the user.
**Functional requirements**

A functional requirement defines a function of the system or one of its components.

- how the system should respond to particular inputs
- how the system should react in particular situations
- may also state what the system should not do

Ideally: each line of code traceable to a functional requirement

**Non-functional requirements**

A non-functional requirement imposes constraints on the design or the implementation.

- often the "-ilities"
- typically cross-cutting or emergent properties:
  - affect whole system: where do you implement usability or reliability?
  - difficult to meet...
  - ... but failure can render entire system useless

**Non-functional requirements**

Non-functional requirements can also constrain

- **process**: how the system is developed
  - "The system shall be implemented in Java SE8."
  - "The system development shall follow the rules in DO-178B."

- **environment**: hardware, software, legal
  - "The system shall run on mobile devices with at least 16GB RAM that run Android KitKat 4.4.4."
  - "The system shall not disclose to the operators of the system any personal information about customers apart from their name and reference number."

**FURPS**

- **Functionality**: capability, ...
- **Usability**: human factors, consistency, documentation, responsiveness, ...
- **Reliability**: availability, recoverability, predictability, accuracy, ...
- **Performance**: speed, efficiency, resource consumption, scalability, ...
- **Supportability**: testability, flexibility, localizability, ...

Standardized way to organize (non-functional) requirements

**Domain requirements**

Application domains shape the system requirements:

- **functional**: complex computations or processes, ...
- **non-functional**: user expectations, standards, ...

**Domain requirements** are often the hardest:

- **understandability**: domain requirements are often expressed in domain-specific terms which are not understood by software engineers
- **implicit**: domain experts often do not think to make the domain requirements explicit
  - they understand the area and requirements well
- **cross-cutting**
Requirements Elicitation

Requirements elicitation techniques
Wide variety of approaches and techniques:
- observational: observe users of existing system
- analytical: analyze documents, forms, processes
- inquisitive: interview users
- speculative: suggest and evaluate functionality
- brainstorming
- rapid prototyping

Usually several methods must be applied.

Stakeholders and viewpoints
A stakeholder is somebody who interacts with or is affected by the system:
- different types of users
- customer
- administrator
- ...

Different stakeholders have different viewpoints on the system, and so different requirements!

Identifying the stakeholders is the first important step in requirements engineering!

Example: meal ordering system

Identify the stakeholders!
**Use case analysis**

*Use case analysis* is a systematic approach to develop requirements, based on use cases:

A use case is a generalized description of one high-level interaction between the system and one or more stakeholders or actors in order to achieve a user goal.

Examples:
- change order
- pay with credit card
- return a book on time
- return a book too late

**Use case diagrams – UML notation**

![Use case diagram](image)

**Example: meal ordering system**

Identify the actors! Identify the use cases!

**Adding detail to use cases**

Short or full descriptions complete use case diagrams:
- **name**: give a short, descriptive name to the use case
- **actors**: list the actors who can perform this use case
- **goals**: explain what the actor is trying to achieve
- **summary**: give a short informal description
- **preconditions**: state of the system before the use case
- **postconditions**: state of the system after completion
- **related use cases**: list generalizations, extensions, ...
- **steps**: describe each step using a 2-column format
  - actor’s actions and system’s responses
  - focus on interaction, not computation

**Use case: Open file by name**

- **actors**: use system
- **steps**: formulate so that different alternatives are allowed
  1. choose “Open…” command
  2. display “File open” dialog
  3. type file name
  4. confirm selection
  5. remove dialog from display
**Use case: Exit car park, paying cash**

- **actors**: car driver
- **goals**: to leave the car park after having paid the amount due in cash
- **summary**: when the car driver wants to leave, s/he must drive the car to the exit barrier and interact with the machine to pay the amount due and leave
- **preconditions**: the car driver must have entered the car park with a car and must have picked up a valid ticket on entry
- **postconditions**: ticket is invalid
- **related use cases**: Exit car park, paying with debit card

**Use case: Exit car park, paying cash**

- **steps**
  - **actor**
  - **system**
  1. drive car to exit barrier, triggering a sensor
  2. detect presence of a car
  3. prompt driver to insert ticket
  4. insert ticket
  5. display amount due
  6. insert money
  7. return change (if any due)
  8. prompt driver to take change (if any due)
  9. raise barrier
  10. drive car through barrier, triggering a sensor
  11. lower barrier

**Use case: Check out a book**

- **actors**
- **goals**
- **summary**
- **preconditions**
- **postconditions**
- **related use cases**
- **steps**

  - **actor**
  - **system**

**Structuring use cases**

For larger systems uses cases should be structured:

- **generalization**: used to represent several similar use cases by a single case
  
  - specializations provide details of the original cases
  
  - cf. superclasses in class diagrams

- **extension**: used to make optional and exceptional cases explicit
  
  - keep the description of the basic use case simple

- **inclusion**: used to factor out commonality between different use cases

  - cf. method calls: avoid repeating details, abstraction

  - Examples: interaction validation, user authorization, ...

**Structuring use cases – example**

**Example: meal ordering system**

[Diagram showing a meal ordering system with steps involving ordering, paying, and leaving]

[Source: T.C. Lehtoivio, R. Lapinainen: Object-Oriented Software Engineering, 2nd ed., 2005]
Requirements Specification

Requirements for requirements
Requirements specifications should be …
• **relevant**: each requirement is pertinent to the problem and its solution
• **traceable**: each requirement can be traced to its origin in the problem environment
• **complete**: the requirements cover all aspects of the user function.
• **prioritized**: each requirement’s importance is given
• **feasible**: each requirement can be realized with the available resources, and within the given constraints
• **testable**: for each requirement it is possible to determine whether it has been satisfied

Requirements specification methods
Requirements can be recorded in different ways:
• **natural language**
• **structured natural language** – standard forms or templates
• **requirements specification languages** – “programming-like”
  – abstract features
  – decision tables
• **graphical notations** (with text annotations)
  – SADT, DFD, ...
  – use cases
• **mathematical specification languages**
  – VDM, Z, B, Event-B, ...
  – automata-, logic-, or set-based

Natural language requirements
Advantages:
• **easy to understand**
  – everybody can read...
• **easy to implement**
  – no specific tools required, Word or LaTeX will do
• **flexible**
  – no constraints on what can be expressed

Example
1. The library information system shall allow users to search the library catalogue for books and audio-visual materials.

Natural language requirements
Disadvantages
• **ambiguity and lack of clarity**
  – synonyms
  – readers must interpret words the same way
  – precision requires longwinded prose
• **over-flexible**
  – not enough structure to enforce uniform style
• **amalgamation and confusion**
  – several requirements may be expressed together
  – functional and non-functional requirements mixed-up
• **lack of modularisation**
2.16.3.f While acting as the bus controller, the C&C MDM CSCI shall set the e, c, w indicators identified in Table 3.2.16-II for the corresponding RT to "failed" and set the failure status to failed for all RT's on the bus upon detection of transaction errors of selected messages to RT's whose 1553 FBIR is not inhibited in two consecutive processing frames within 100 milliseconds of detection of the second transaction error if a backup BC is available, the BC has been switched in the last 20 seconds, the SPD card reset capability is inhibited, or the SPD card has been reset in the last 10 major (10 second) frames, and either 1. the transaction errors are from multiple RT's, the current channel has been reset within the last major frame, or 2. the transaction errors are from multiple RT's, the bus channel's reset capability is inhibited, and the current channel has not been reset within the last major frame.

### Structured Natural Language

#### Idea: limit flexibility of natural language with forms or templates in order to increase uniformity

- easier to read and to check completeness

Typical format (cf. use cases):

- definition of the function or entity
- description of inputs and where they come from
- description of outputs and where they go to
- other entities used
- description of the action to be taken
- pre- and post-conditions (if appropriate)
- side effects (if any)

#### Table: Tabular specifications

- condition/action tables

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sugar level falling (o2 &lt; 1)</td>
<td>CompDose = 0</td>
</tr>
<tr>
<td>Sugar level stable (o2 = 1)</td>
<td>CompDose = 0</td>
</tr>
<tr>
<td>Sugar level increasing and rate of increase decreasing (o2 &gt; 1) x (1 -&gt; 0)</td>
<td>CompDose = 0</td>
</tr>
<tr>
<td>Sugar level increasing and rate of increase increasing (o2 &gt; 1) x (1 -&gt; 0)</td>
<td>CompDose = round [(o2 - 1)/4]</td>
</tr>
<tr>
<td>Sugar level increasing and rate of increase stable or increasing (o2 &gt; 1) x (1 -&gt; 0)</td>
<td>CompDose = MinimumDose</td>
</tr>
</tbody>
</table>

- useful for systems with a number of possible alternative courses of action
- particularly control systems
Writing Good Requirements

Bad requirements
Many projects suffer from...
• poorly structured requirements documents
• poorly written individual requirements
• poorly validated or untestable requirements
• poor customer involvement
• poor prioritization
  – analysis paralysis, scope creep, gold plating
• poor change management
  – entropic decay

Templates are Essential
• define standard requirements templates
  – easy to find information
  – nothing will be “forgotten”
• define a standard document structure
  – readers familiar with the document
  – acts as a checklist so that no sections are forgotten

IEEE requirements document template
1 Introduction
  1.1 Purpose
  1.2 Scope
  1.3 Definitions, acronyms, and abbreviations
  1.4 References
  1.5 Overview
2 Overall Description
  2.1 Product perspective
  2.2 Product functions
  2.3 User characteristics
  2.4 Constraints
  2.5 Assumptions and Dependences
3 Specific Requirements
Appendices
Glossary
Index

Specific Requirements
How do we organize Section 3 (Specific requirements)?
• depends on the system under consideration
• organized to provide
  – clarity
  – changeability
  – etc.
• 8 examples suggested in IEEE Standard
3 Specific Requirements

3.1 External interface requirements
  3.1.1 User interfaces
  3.1.2 Hardware interfaces
  3.1.3 Software interfaces
  3.1.4 Communications interfaces

3.2 Functional requirements

3.3 Performance requirements

3.4 Design constraints

3.5 Software system attributes

3.6 Other requirements

Specific Requirements by Mode

3 Specific Requirements

3.2 Functional requirements
  3.2.1 Mode 1
    3.2.1.1 Requirement 1.1
    3.2.1.2 Requirement 1.2
    3.2.1.3 Requirement 1.3
  3.2.2 Mode 2

3.3 Performance

3.4 Design constraints

3.5 Software system attributes

3.6 Other requirements

Specific Requirements by User class

3 Specific Requirements

3.2 Functional requirements
  3.2.1 User class 1
    3.2.1.1 Requirement 1.1
    3.2.1.2 Requirement 1.2
    3.2.1.3 Requirement 1.3
  3.2.2 User class 2
    3.2.2.1 Requirement 2.1
    3.2.2.2 Requirement 2.2

3.3 Performance

3.4 Design constraints

3.5 Software system attributes

3.6 Other requirements

Specific Requirements by Feature

3 Specific Requirements

3.2 System features
  3.2.1.1 System feature 1
    3.2.1.1.1 Purpose of feature
    3.2.1.1.2 Stimulus/response sequence
  3.2.1.3 System feature 3

3.3 Performance

3.4 Design constraints

3.5 Software system attributes

3.6 Other requirements

Requirements documents should NOT include...

- project development plans
  - cost, staffing, schedules, methods, tools, etc.
  - lifetime of SRS is entire operational life of the product
  - lifetime of development plans is much shorter
- product assurance plans
  - configuration management, V&V, test, QA, etc.
  - different audiences and timelines
- design considerations
  - except if the application domain constrains the design
    - e.g., limited bandwidth or security concerns
  - focus on what, not on how

Writing style

Requirements are technical documents: brevity, clarity, consistency, and precision are more important than literary style!

- avoid requirements “fusion”
  - one functionality per requirement
- use consistent terminology
  - define terms in glossary
- BE PRECISE – no vague requirements
Writing style: BE PRECISE

- avoid persuasive words
  - certainly, therefore, clearly, obviously, ...
- avoid vague words
  - some, most, sometimes, often, usually, normally, ordinarily, typically, customarily, ...
- avoid non-committal words
  - ought to, preferred, desirable, wanted, ...
- avoid and/or, etc. like the plague
- avoid generic verbs
  - handled, processed, rejected, skipped, ...
- avoid unquantifiable words
  - reduce, optimize, large, rapid, user-friendly, easy, simple, intuitive, robust, efficient, flexible, ...

Writing Style: SHALL – SHOULD – MAY

Use signifiers to denote the urgency of a requirement:

- SHALL means that the definition is an absolute requirement of the specification (i.e., essential)
- SHOULD means that there may exist valid reasons in particular circumstances to ignore a particular requirement (i.e., desirable)
- MAY means that a requirement is truly optional

More information:
S. Brauer: Key words for use in RFCs to Indicate Requirement Levels. IETF RFC 2119, 1997

Example: ATM

2.6 Withdrawal

If the card is accepted, the user has entered the correct PIN, and there are sufficient funds in the account, the amount of cash shall be dispensed. If the card is invalid (in which case it should be ejected), the PIN does not match the one required for the card (in which case a tone shall sound and the user given the option to try again—the tries shall be limited to 3), or the balance is insufficient (in which case a tone shall sound and the user shall have the opportunity to enter a new amount) cash shall not be dispensed.

Example: ATM – restructured

2.6: The System shall support cash withdrawals by the user.

2.6.1: A withdrawal shall be allowed if and only if:
   a. The card can be validated (Req. 2.7).
   b. The PIN is valid for the card (Req 2.8).
   c. The funds in the card account exceeds the funds requested in the withdrawal.

2.6.2: If a withdrawal is allowed (2.6.1), the exact amount requested shall be dispensed.

Customer involvement

Lack of user input is major (1 in 8) cause of project failures

Warning sign: user surrogates (user managers, marketing staff) supply all the requirements
- identify the various user classes
  - not all users are the same
- find “product champions” for specific user classes

Scope creep

scope creep: new requirements are continually added
- define the project scope clearly
  - expect some requirements growth in all projects
  - include buffers in project plan
- attach a person to each requirement
  - people are much less likely to add “the kitchen sink” if their name is there
**Analysis paralysis and gold plating**

- **analysis paralysis**: requirements development seems to go on forever
- **gold plating**: developers add unnecessary functionality
  - identify key decision-makers to “call the shots”
  - prioritize requirements
  - trace requirements to use cases

**Change management**

- **plan for change**:
  - use version control system for requirements
  - change impact analysis
    - identify related requirements in document

**Requirements validation techniques**

- **Goal**: check requirements for realism, consistency, and completeness

- **prototyping**
  - use executable system model to check requirements
- **reviews**
  - systematic manual analysis of the requirements.
- **test-case generation**
  - develop tests for requirements to check testability: “If you can’t test it, it is not a requirement!”

**Requirements Validation**

**Review checks**

- **comprehensibility**: is the requirement properly understood?
- **traceability**: is the origin of the requirement clearly stated?
- **adaptability**: can the requirement be changed without a large impact on other requirements?
  - easier if the document is properly cross-referenced
- **verifiability**: is the requirement realistically testable?

**Traceability**

**Requirements traceability** is the linking between requirements and other artefacts:

- **use cases**
  - system validation (do we plan the right system?)
- **other requirements**
  - cross-referencing
  - user-to-system: capture hierarchical structure
- **code**
  - system validation (did we implement the right system?)
- **test cases**
  - verification (did we implement the system right?)
Traceability techniques

- assign a unique number to all requirements and other items
- cross-reference using this unique number
- representation typically as traceability matrix
  - several matrices for different types of traceability links
- main problems
  - link maintenance
  - link recovery

Requirements verifiability

- requirements must be verifiable, i.e., written so that their validity in the system can be checked by one or many test case(s)
  "If you can't test it, it is not a requirement!"
- even more difficult for non-functional requirements

For example

4. The system should be easy to use by experienced engineers and should be organized in such a way that user errors are minimized.

- How do you test "easy-to-use"?
- What is minimized??

Requirements verifiability

4.1 Engineering staff with more than two years of professional experience shall be able to use all the system functions after a total of two hours training.

4.2 After this training, the average number of errors made by experienced engineers shall not exceed two per day.

Requirements verifiability

Many non-functional properties can be quantified:

- speed processed transactions/second
- user/event response time
- screen refresh time
- size
  - minimum ram requirement
  - database size
- ease of use
  - average user training time
  - number of help frames
- reliability
  - mean time to failure
  - probability of unavailability
  - rate of failure occurrence
  - availability

Requirements verifiability

Many non-functional properties can be quantified:

- robustness time to restart after failure
- percentage of events causing failure
- probability of data corruption on failure

- portability % of target dependent statements
- number of target systems

Summary

- requirements capture what a system shall do
  - but avoid design detail as much as possible
  - written in the user's language (with all the problems...)
- requirements are difficult to find out and formalize
- requirements problems are
  - common
  - the most costly
  - the most difficult to correct (they are conceptual)
- use a standard structure, forms, and checklists for the document and the individual requirements
- write verifiable requirements
Poor requirements are the source of all evil!