Softwaretechnik / Software-Engineering

Lecture 6: Requirements Engineering

2016-05-12

Prof. Dr. Andreas Podelski, **Dr. Bernd Westphal**

Albert-Ludwigs-Universität Freiburg, Germany

You Are Here.

Introduction	L 1:	18.4., Mon
Scales, Metrics, Costs	L 2:	21.4., Thu
	L 3:	25.4., Mon
	T 1:	28.4., Thu
Development	L 4:	2.5., Mon
	-	5.5., Thu
Process	L 5:	9.5., Mon
	L 6:	12.5., Thu
	-	16.5., Mon
	-	19.5., Thu
	T 2:	23.5., Mon
	-	26.5., Thu
Requirements Engineering	L 7:	30.5., Mon
	L 8:	2.6., Thu
	L 9:	6.6., Mon
	T 3:	9.6., Thu
Architecture & Design	L10:	13.6., Mon
	L 11:	16.6., Thu
	L12:	20.6., Mon
	T 4:	23.6., Thu
Software Mondelling	L13:	27.6., Mon
	L14:	30.6., Thu
	L15:	4.7., Mon
	T 5:	7.7., Thu
Quality Assurance (Testing, Formal Verification)	L16:	11.7., Mon
	L17:	14.7., Thu
	L18:	18.7., Mon
Wrap-Up	L19:	21.7., Thu

Topic Area Requirements Engineering: Content

VL 6

- **Introduction**
- **Requirements Specification**
  - Desired Properties
  - Kinds of Requirements
  - Analysis Techniques

⋮

- **Documents**
  - Dictionary, Specification

vocabularity, concepts

- **Specification Languages**
  - Natural Language — informal
  - Working Definition: Software

VL 7

  - Decision Tables — formal
    - Syntax, Semantics

⋮

    - Consistency, Completeness, ...

VL 8

  - Scenarios
    - User Stories, Use Cases — semi-formal

⋮

    - Live Sequence Charts — formal
      - Syntax, Semantics

VL 9

⋮

- **Discussion**

Recall: Structure of Topic Areas

Example: Requirements Engineering

Vocabulary

e.g. consistent, complete, tacit, etc.

Techniques

informal

↕

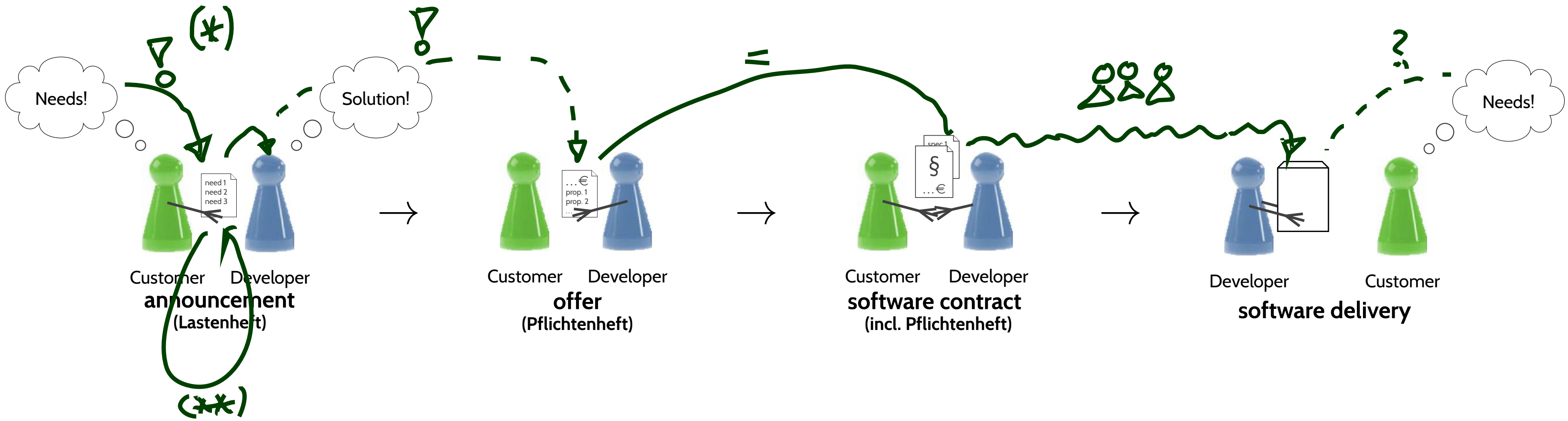
semi-formal

↕

formal

Content

Introduction



requirement –

(1) A condition or capability needed by a user to solve a problem or achieve an objective.

(2) 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 documents.

(3) A documented representation of a condition or capability as in (1) or (2).

IEEE 610.12 (1990)

requirements analysis –

(1) The process of studying user needs to arrive at a definition of system, hardware, or software requirements. (*)

(2) The process of studying and refining system, hardware, or software requirements. (**)

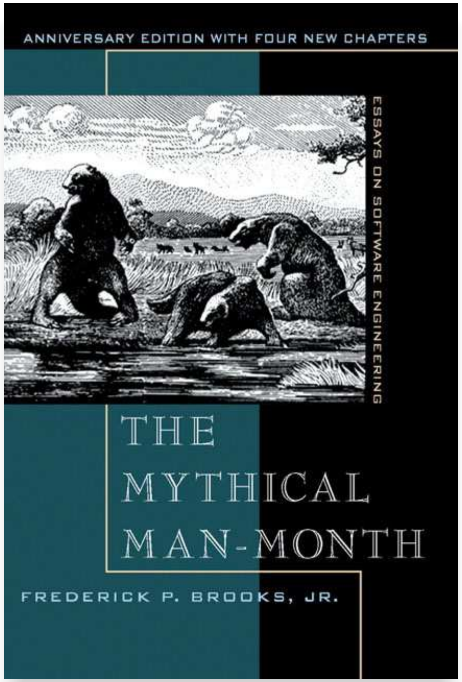
IEEE 610.12 (1990)

The hardest single part of building a software system is deciding precisely what to build.

No other part of the conceptual work is as difficult as establishing the detailed technical requirements ...

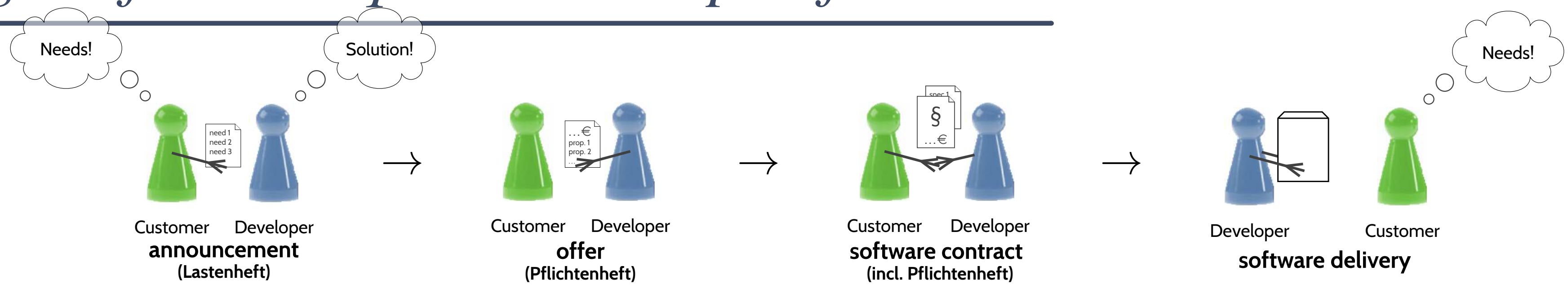
No other part of the work so cripples the resulting system if done wrong.

No other part is as difficult to rectify later. F.P. Brooks (Brooks, 1995)



– 6 – 2016-05-12 – Sreintro –

Usages of The Requirements Specification



- **negotiation**
  (with customer, marketing department, or ...)
- **design** and **implementation**,
  - without specification, programmers may just "ask around" when in doubt, possibly yielding different interpretations → **difficult integration**
- **documentation**, e.g., the **user's manual**,
  - without specification, the user's manual author can only describe what the system **does**, not what it should do (**"every observation is a feature"**)
- preparation of **tests**,
  - without a description of allowed outcomes, tests are randomly searching for generic errors (like crashes) → **systematic testing hardly possible**

- **acceptance** by customer,
  resolving later objections or regress claims,
  - without specification, it is unclear at delivery time whether behaviour is an error (developer needs to fix) or correct (customer needs to accept and pay) → **nasty disputes**, **additional effort**
- **re-use**,
  - without specification, re-use needs to be based on re-reading the code → **risk of unexpected changes**
- later **re-implementations**.
  - the new software may need to adhere to requirements of the old software; if not properly specified, the new software needs to be a 1:1 re-implementation of the old → **additional effort**

Requirements Specifications

Requirements Analysis...

... in the sense of **"finding out what the exact requirements are"**.

"Analysing an existing requirements/feature specification" $\rightarrow$ later.

In the following we shall discuss:

(i) desired **properties** of

- requirements specifications,
- requirements specification documents,

(ii) **kinds** of requirements

- hard and soft,
- open and tacit,
- functional and non-functional.

(iii) (a selection of) **analysis techniques**

(iv) **documents** of the requirements analysis:

- dictionary,
- requirements specification ('Lastenheft'),
- feature specification ('Pflichtenheft').

- **Note**: In the following (unless otherwise noted), we discuss the **feature specification**, i.e. the document on which the software development is based.

  To maximise confusion, we may occasionally (inconsistently) call it **requirements specification** or just **specification** – should be clear from context...

- **Recall**: one and the same content can serve both purposes; only the title defines the purpose then.

Requirements on Requirements Specifications

A **requirements specification** should be

- **correct**
  – it correctly represents the wishes/needs of the customer,
- **complete**
  – all requirements (existing in somebody's head, or a document, or …) should be present,
- **relevant**
  – things which are not relevant to the project should not be constrained,
- **consistent**, **free of contradictions**
  – each requirement is compatible with all other requirements; otherwise the requirements are **not realisable**,
- **neutral**, **abstract**
  – a requirements specification does not constrain the realisation more than necessary,
- **traceable**, **comprehensible**
  – the sources of requirements are documented, requirements are uniquely identifiable,
- **testable**, **objective**
  – the final product can **objectively** be checked for satisfying a requirement.

- **Correctness** and **completeness** are defined **relative** to something which is usually only **in the customer's head**.

  → is is **difficult** to **be sure of correctness** and **completeness**.

- **"Dear customer, please tell me what is in your head!"** is in almost all cases **not a solution**!

  It's not unusual that even the customer does not precisely know…!

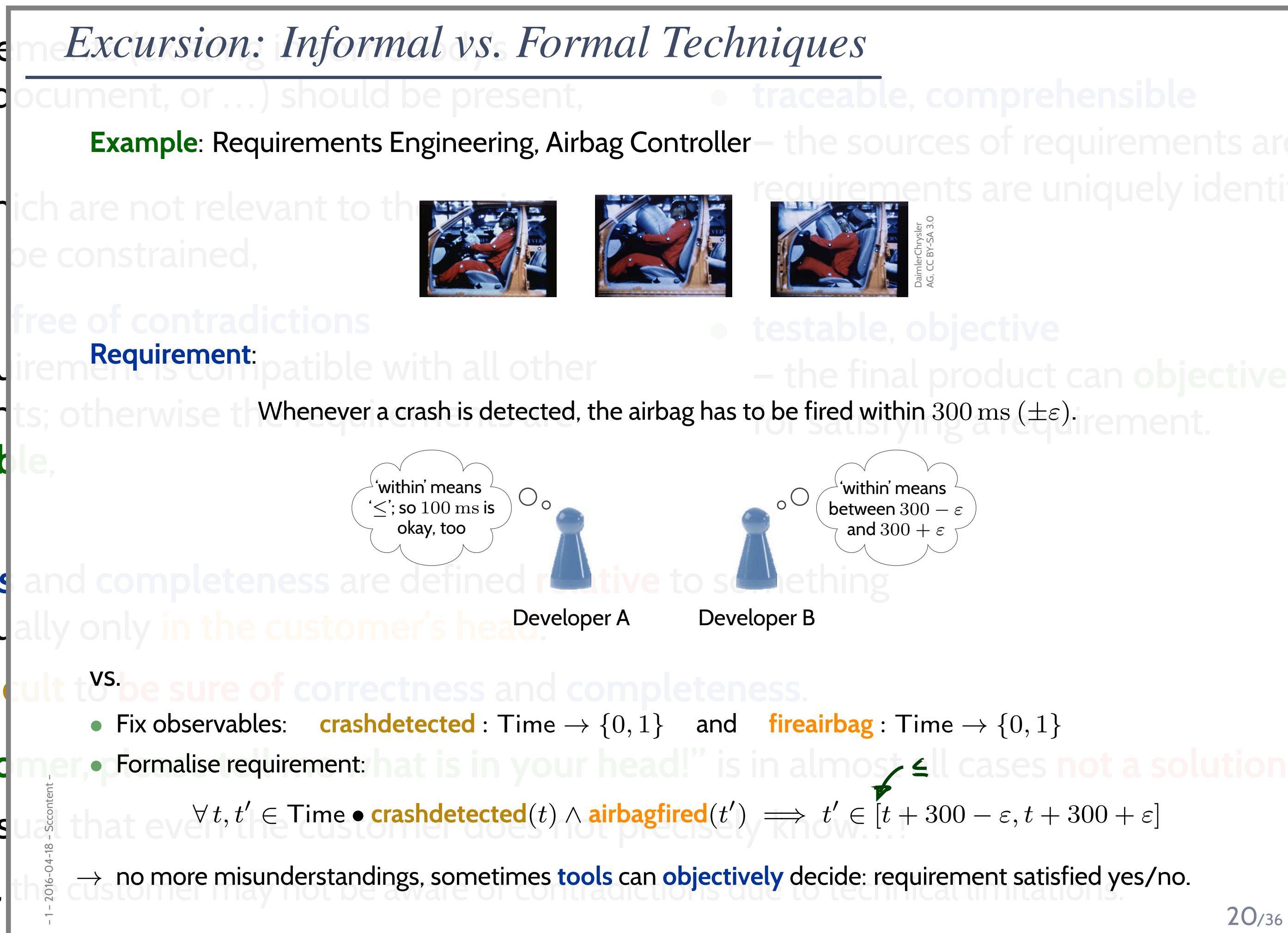
  For example, the customer may not be aware of contradictions due to technical limitations.

Requirements on Requirements Specifications

A **requirements specification** should be

- **correct**
  – it correctly represents the wishes/needs of the customer,
- **complete**
  – all requirements (existing in somebody's head, or a document, or ...) should be present,
- **relevant**
  – things which are not relevant to the
  should not be constrained,
- **consistent**, free of contradictions
  – each requirement is compatible with all other requirements; otherwise the requirements are
  not realisable,

- **neutral**, **abstract**
  – a requirements specification does not constrain the realisation more than necessary,
- traceable, comprehensible
  – the sources of requirements are documented, requirements are uniquely identifiable,
- testable, objective
  – the final product can objectively be checked for satisfying a requirement.



Excursion: Informal vs. Formal Techniques

Example: Requirements Engineering, Airbag Controller

DaimlerChrysler AG, CC BY-SA 3.0

Requirement:

Whenever a crash is detected, the airbag has to be fired within $300\,\mathrm{ms}$ $(\pm\varepsilon)$.

'within' means '$\leq$'; so $100\,\mathrm{ms}$ is okay, too

'within' means between $300-\varepsilon$ and $300+\varepsilon$

Developer A

Developer B

vs.

- Fix observables: crashdetected : Time $\to \{0,1\}$ and fireairbag : Time $\to \{0,1\}$
- Formalise requirement:

$$\forall\, t, t' \in \text{Time} \bullet \text{crashdetected}(t) \wedge \text{airbagfired}(t') \implies t' \in [t+300-\varepsilon, t+300+\varepsilon]$$

$\leq$

$\to$ no more misunderstandings, sometimes **tools** can **objectively** decide: requirement satisfied yes/no.

– 1 – 2016-04-18 – Scontent –

20/36

- **Correctness** and **completeness** are defined **relative** to something which is usually only **in the customer's head**.

  $\longrightarrow$ is is **difficult** to **be sure of correctness** and **completeness**.
- **"Dear customer, please tell me what is in your head!"** is in almost all cases **not a solution!**

  It's not unusual that even the customer does not precisely know...!

  For example, the customer may not be aware of contradictions due to technical limitations.

Requirements on Requirements Specification Documents

The **representation** and **form** of a requirements specification should be:

- **easily understandable**,
  not unnecessarily complicated –
  all affected people should be able to understand the requirements specification,
- **precise** –
  the requirements specification should not introduce new unclarities or rooms for interpretation ($\rightarrow$ testable, objective),
- **easily maintainable** –
  creating and maintaining the requirements specification should be easy and should not need unnecessary effort,
- **easily usable** –
  storage of and access to the requirements specification should not need significant effort.

Note: Once again, it's about compromises.

- A very precise **objective** requirements specification may not be easily understandable by every affected person.

  $\rightarrow$ provide redundant explanations.

- It is not trivial to have both, low maintenance effort and low access effort.

  $\rightarrow$ **value low access effort higher**,
  a requirements specification document is much more often **read** than **changed** or **written** (and most changes require reading beforehand).

Pitfall: Vagueness vs. Abstraction

Consider the following examples:

- **Vague** (not precise):

  "the list of participants should be sorted conveniently"

- **Precise**, abstract:

  "the list of participants should be sorted by immatriculation number, lowest number first"

- **Precise**, non-abstract:

  "the list of participants should be sorted by

```
public static <T> void Collections::sort( List<T> list, Comparator c );
```

  where T is the type of participant records, c compares immatriculation number numerically."

- A requirements specification should always be as **precise** as possible ($\rightarrow$ testable, objective).

  It need not denote **exactly one solution**;
  precisely characterising acceptable solutions is often more appropriate.

- Being to(o) specific, may limit the design decisions of the developers, which may cause unnecessary costs.

- Idealised views advocate a strict **separation** between
  requirements ("what is to be done?") and **design** ("how are things to be done?").

Kinds of Requirements

Kinds of Requirements: Functional and Non-Functional

- **Proposal**: View software S as a **function**

$$S : i_1, i_2, i_3, \dots \mapsto o_0, o_1, o_2, \dots$$

  which maps **sequences of inputs** to **sequences of outputs**.

  Examples:

  - Software "compute shipping costs":
    - o_0: initial state,
    - i_1: shipping parameters (weight, size, destination, ...),
    - o_1: shipping costs

    And no more inputs, $S : i_1 \mapsto o_1$.

  - Software "traffic lights controller":
    - o_0: initial state,
    - i_1: pedestrian presses button,
    - $o_1, o_2, \dots$: stop traffic, give green to pedestrians,
    - i_n: button pushed again
    - ...

- **Every constraint** on things which are **observable** in the sequences is a **functional requirement** (because it requires something for the function S).

  Thus **timing**, **energy consumption**, etc. may be subject to functional requirements.

- Clearly **non-functional** requirements:

  programming language, coding conventions, process model requirements, portability...

Kinds of Requirements: Hard and Soft Requirements

- **Example** of a **hard requirement**:
  - Cashing a cheque over N € must result in a new balance decreased by N; there is not a micro-cent of tolerance.
- **Examples** of **soft requirements**:
  - If a vending machine dispenses the selected item within 1 s, it's clearly fine; if it takes 5 min., it's clearly wrong – where's the boundary?
  - A car entertainment system which produces "noise" (due to limited bus bandwidth or CPU power) in average once per hour is acceptable, once per minute is not acceptable.

The border between hard/soft **is difficult to draw**, and

- as **developer**, we want requirements specifications to be "**as hard as possible**", i.e. we want a clear right/wrong.
- as **customer**, we often cannot provide this clarity; we know what is "**clearly wrong**" and we know what is "**clearly right**", but we don't have a sharp boundary.

$\rightarrow$ intervals, rates, etc. can serve as **precise specifications** of **soft requirements**.

Kinds of Requirements: Open and Tacit

- **open**: customer is aware of and able to explicitly communicate the requirement,
- **(semi-)tacit**:
  customer not aware of something **being** a requirement (obvious to the customer but not considered relevant by the customer, not known to be relevant).

  Examples:

  - buttons and screen of a mobile phone should be on the same side,
  - important web-shop items should be on the right hand side because the main users are socialised with right-to-left reading direction,
  - the ECU (embedded control unit) may only be allowed use a certain amount of bus capacity.

Customer/Client	**Analyst**: knows domain	**Analyst**: new to domain
explicit	requirements discovered	requirements discoverable
semi-tacit	requirements discoverable	requirements discoverable with difficulties
tacit	hard/impossible to discover	

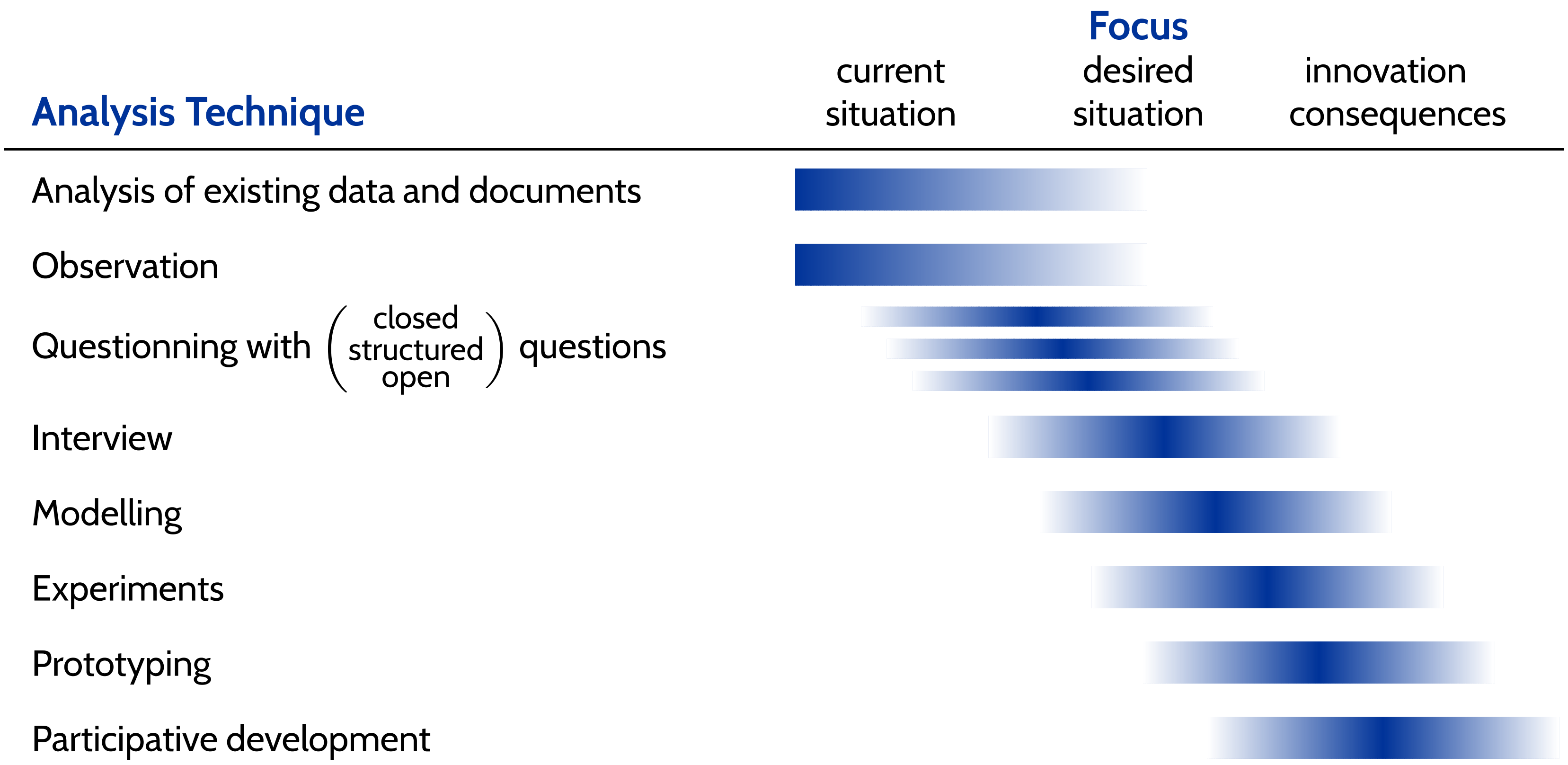
(Gacitua et al., 2009)

- distinguish **don't care**:

  intentionally left open to be decided by developer.

Requirements Analysis Techniques

(A Selection of) Analysis Techniques



(Ludewig and Lichter, 2013)

Requirements Elicitation

- **Observation**:

  Customers can not be assumed to be trained in stating/communicating requirements.

- It is the **task of the analyst** to:
  - **ask** what is wanted,
    ask what is not wanted,
  - establish **precision**,
    look out for contradictions,
  - **anticipate** exceptions, difficulties, corner-cases,
  - have technical background to **know** technical difficulties,
  - **communicate** (formal) specification to customer,
  - "test" own understanding by **asking more** questions.

  → i.e. to **elicit** the requirements.

Goal: automate opening/closing of a main door with a new software.

A **made up** dialogue…

Analyst: *So in the morning, you open the door at the main entrance?*
 Customer: *Yes, as I told you.*
A: *Every morning?*
 C: *Of course.*
A: *Also on the weekends?*
 C: *No, on weekends, the entrance stays closed.*
A: *And during company holidays?*
 C: *Then it also remains closed of course.*
A: *And if you are ill or on vacation?*
 C: *Then Mr. M opens the door.*
A: *And if Mr. M is not available, too?*
 C: *Then the first client will knock on the window.*
A: *Okay. Now what exactly does "morning" mean?*
 …

(Ludewig and Lichter, 2013)

How Can Requirements Engineering Look In Practice?

- Set up a **core team** for analysis (3 to 4 people), include experts from the **domain** and **developers**. Analysis benefits from **highest skills** and **strong experience**.

- During analysis, talk to **decision makers** (managers), domain **experts**, and **users**.

  Users can be interviewed by a team of 2 analysts, ca. 90 min.

- The resulting "**raw material**" is sorted and assessed in half- or full-day workshops in a team of 6-10 people.

  Search for, e.g., **contradictions** between customer wishes, and for **priorisation**.

  Note: **The customer decides**. Analysts may make **proposals** (different options to choose from), but the customer chooses. (And the choice is documented.)

- The "raw material" is basis of a **preliminary requirements specification** (audience: the developers) with open questions.

  Analysts need to **communicate** the requirements specification **appropriately** (explain, give examples, point out particular corner-cases).

  Customers without strong maths/computer science background are often **overstrained** when "left alone" with a **formal** requirements specification.

- **Result**: **dictionary**, **specified requirements**.

- Many customers do not want **(radical) change**, but **improvement**.
- Good questions: How are things done today? What should be improved?

Requirements Documents

Dictionary

- Requirements analysis should be based on a **dictionary**.
- A **dictionary** comprises definitions and clarifications of **terms** that are relevant to the project and of which different people (in particular customer and developer) may have different understandings before agreeing on the dictionary.
- Each **entry** in the **dictionary** should provide the following information:
  - **term** and **synonyms** (in the sense of the requirements specification),
  - **meaning** (definition, explanation),
  - **deliminations** (where **not** to use this terms),
  - **validness** (in time, in space, ...),
  - **denotation**, unique identifiers, ...,
  - **open questions** not yet resolved, !
  - **related terms**, cross references.

  Note: entries for terms that **seemed** "crystal clear" at first sight are **not uncommon**.

- All work on requirements should, as far as possible, be done **using terms from the dictionary** consistently and consequently.

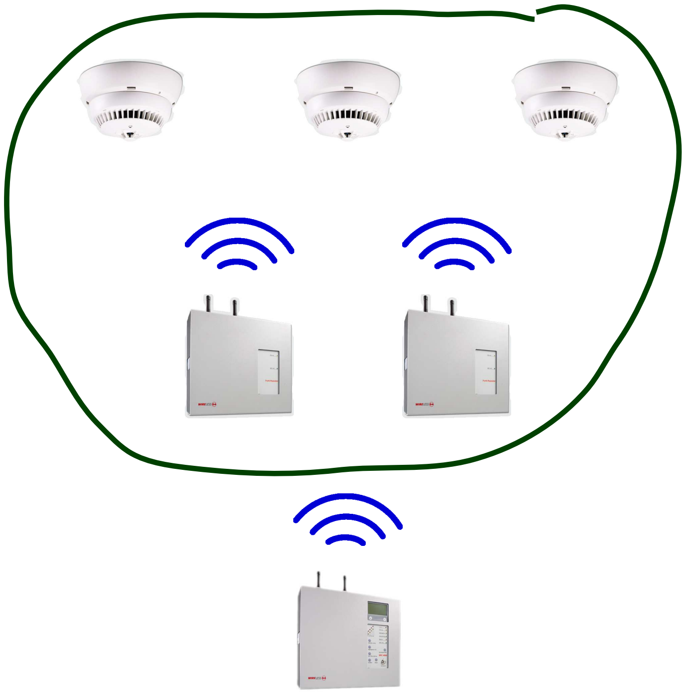
  The dictionary should in particular be **negotiated with the customer** and used in communication (if not possible, at least developers should stick to dictionary terms).

- **Note**: do not mix up **real-world/domain** terms with ones only "living" in the software.

Dictionary Example

Example: Wireless Fire Alarm System

- During a project on designing a highly reliable, EN-54-25 conforming wireless communication protocol, we had to learn that the relevant components of a fire alarm system are
  - **terminal participants** (heat/smoke sensors and manual indicators),
  - **repeaters** (a non-terminal participant),
  - and **a central unit** (not a participant).
- Repeaters and central unit are technically very similar, but need to be distinguished to understand requirements. The **dictionary** explains these terms.



(Arenis et al., 2014)

Excerpt from the dictionary (ca. 50 entries in total):

Part A part of a fire alarm system is either a **participant** or a **central unit**.

Repeater A repeater is a **participant** which accepts messages for the **central unit** from other **participants**, or messages from the **central unit** to other **participants**.

Central Unit A central unit is a **part** which receives messages from different assigned **participants**, assesses the messages, and reacts, e.g. by forwarding to persons or optical/acustic signalling devices.

Terminal Participant A terminal participant is a **participant** which is not a **repeater**. Each terminal participant consists of exactly one wireless communication module and devices which provide sensor and/or signalling functionality.

Requirements Specification

specification – A document that specifies,

- in a complete, precise, verifiable manner,

the

- requirements, design, behavior, or other characteristics of a system or component,

and, often, the procedures for determining whether these provisions have been satisfied. **IEEE 610.12 (1990)**

software requirements specification (SRS) – Documentation of the essential requirements (functions, performance, design constraints, and attributes) of the software and its external interfaces. **IEEE 610.12 (1990)**

IEEE Std 830-1998
(Revision of
IEEE Std 830-1993)

IEEE Recommended Practice for Software Requirements Specifications

Sponsor

Software Engineering Standards Committee
of the
IEEE Computer Society

Approved 25 June 1998

IEEE-SA Standards Board

Abstract: The content and qualities of a good software requirements specification (SRS) are described and several sample SRS outlines are presented. This recommended practice is aimed at specifying requirements of software to be developed but also can be applied to assist in the selection of in-house and commercial software products. Guidelines for compliance with IEEE/EIA 12207.1-1997 are also provided.
Keywords: contract, customer, prototyping, software requirements specification, supplier, system requirements specifications

---

The Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street, New York, NY 10017-2394, USA

ISBN 0-7381-0332-2

Structure of a Requirements Document: Example

1 INTRODUCTION

- 1.1 Purpose
- 1.2 Acronyms and Definitions] Dictionary
- 1.3 References
- 1.4 User Characteristics

2 FUNCTIONAL REQUIREMENTS

- 2.1 Function Set 1
- 2.2 etc.

3 REQUIREMENTS TO EXTERNAL INTERFACES

- 3.1 User Interfaces
- 3.2 Interfaces to Hardware
- 3.3 Interfaces to Software Products / Software / Firmware
- 3.4 Communication Interfaces

4 REQUIREMENTS REGARDING TECHNICAL DATA

- 4.1 Volume Requirements
- 4.2 Performance
- 4.3 etc.

5 GENERAL CONSTRAINTS AND REQUIREMENTS

- 5.1 Standards and Regulations
- 5.2 Strategic Constraints
- 5.3 Hardware
- 5.4 Software
- 5.5 Compatibility
- 5.6 Cost Constraints
- 5.7 Time Constraints
- 5.8 etc.

6 PRODUCT QUALITY REQUIREMENTS

- 6.1 Availability, Reliability, Robustness
- 6.2 Security
- 6.3 Maintainability
- 6.4 Portability
- 6.5 etc.

7 FURTHER REQUIREMENTS

- 7.1 System Operation
- 7.2 Customisation
- 7.3 Requirements of Internal Users

(Ludewig and Lichter, 2013) based on (IEEE, 1998)

Content

Specification Languages

Requirements Specification Language

specification language – A language, often a machine-processible combination of natural and formal language, used to express the requirements, design, behavior, or other characteristics of a system or component.

For example, a design language or requirements specification language. Contrast with: programming language; query language. **IEEE 610.12 (1990)**

requirements specification language – A specification language with special constructs and, sometimes, verification protocols, used to develop, analyze, and document hardware or software requirements. **IEEE 610.12 (1990)**

Natural Language Specification (Ludewig and Lichter, 2013) based on (Rupp and die SOPHISTen, 2009)

	rule	explanation, example
R1	State each requirement in **active voice**.	Name the actors, indicate whether the user or the system does something. Not “the item is deleted”.
R2	Express processes by **full verbs**.	Not “is”, “has”, but “reads”, “creates”; full verbs require information which describe the process more precisely. Not “when data is consistent” but “after program P has checked consistency of the data”.
R3	Discover **incompletely defined verbs**.	In “the component raises an error”, ask whom the message is addressed to.
R4	Discover **incomplete conditions**.	Conditions of the form “if-else” need descriptions of the if- and the then-case.
R5	Discover **universal quantifiers**.	Are sentences with “never”, “always”, “each”, “any”, “all” really universally valid? Are “all” really all or are there exceptions.
R6	Check **nominalisations**.	Nouns like “registration” often hide complex processes that need more detailed descriptions; the verb “register” raises appropriate questions: who, where, for what?
R7	Recognise and refine **unclear substantives**.	Is the substantive used as a generic term or does it denote something specific? Is “user” generic or is a member of a specific classes meant?
R8	Clarify **responsibilities**.	If the specification says that something is “possible”, “impossible”, or “may”, “should”, “must” happen, clarify who is enforcing or prohibiting the behaviour.
R9	Identify **implicit assumptions**.	Terms (“the firewall”) that are not explained further often hint to implicit assumptions (here: there seems to be a firewall).

Natural Language Patterns

Natural language requirements can be (tried to be) written as an instance of the **pattern** "$\langle A \rangle$ $\langle B \rangle$ $\langle C \rangle$ $\langle D \rangle$ $\langle E \rangle$ $\langle F \rangle$." (German grammar) where

A	clarifies when and under what conditions the activity takes place
B	is MUST (obligation), SHOULD (wish), or WILL (intention); also: MUST NOT (forbidden)
C	is either "the system" or the concrete name of a (sub-)system
D	one of three possibilities: • "does", description of a system activity, • "offers", description of a function offered by the system to somebody, • "is able if", usage of a function offered by a third party, under certain conditions
E	extensions, in particular an object
F	the actual process word (what happens)

(Rupp and die SOPHISTen, 2009)

Example:

After office hours ($= A$), the system ($= C$) should ($= B$) offer to the operator ($= D$) a backup ($= F$) of all new registrations to an external medium ($= E$).

Other Pattern Example: RFC 2119

```
Network Working Group                                         S. Bradner
Request for Comments: 2119                            Harvard University
BCP: 14                                                       March 1997
Category: Best Current Practice


        Key words for use in RFCs to Indicate Requirement Levels

Status of this Memo

   This document specifies an Internet Best Current Practices for the
   Internet Community, and requests discussion and suggestions for
   improvements.  Distribution of this memo is unlimited.

Abstract

   In many standards track documents several words are used to signify
   the requirements in the specification.  These words are often
   capitalized.  This document defines these words as they should be
   interpreted in IETF documents.  Authors who follow these guidelines
   should incorporate this phrase near the beginning of their document:

      The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL
      NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED",  "MAY", and
      "OPTIONAL" in this document are to be interpreted as described in
      RFC 2119.

   Note that the force of these words is modified by the requirement
   level of the document in which they are used.

1. MUST   This word, or the terms "REQUIRED" or "SHALL", mean that the
   definition is an absolute requirement of the specification.

2. MUST NOT   This phrase, or the phrase "SHALL NOT", mean that the
   definition is an absolute prohibition of the specification.

3. SHOULD   This word, or the adjective "RECOMMENDED", mean that there
   may exist valid reasons in particular circumstances to ignore a
   particular item, but the full implications must be understood and
   carefully weighed before choosing a different course.

4. SHOULD NOT   This phrase, or the phrase "NOT RECOMMENDED" mean that
   there may exist valid reasons in particular circumstances when the
   particular behavior is acceptable or even useful, but the full
   implications should be understood and the case carefully weighed
   before implementing any behavior described with this label.



Bradner                   Best Current Practice                 [Page 1]
```

```
RFC 2119                     RFC Key Words

5. MAY   This word, or the adjective "OPTIONAL", mean that a
   truly optional.  One vendor may choose to include the ite
   particular marketplace requires it or because the vendor
   it enhances the product while another vendor may omit the
   An implementation which does not include a particular opt
   prepared to interoperate with another implementation whic
   include the option, though perhaps with reduced functiona
   same vein an implementation which does include a particul
   MUST be prepared to interoperate with another implementat
   does not include the option (except, of course, for the f
   option provides.)

6. Guidance in the use of these Imperatives

   Imperatives of the type defined in this memo must be used
   and sparingly.  In particular, they MUST only be used whe
   actually required for interoperation or to limit behavior
   potential for causing harm (e.g., limiting retransmisssio
   example, they must not be used to try to impose a particu
   on implementors where the method is not required for
   interoperability.

7. Security Considerations

   These terms are frequently used to specify behavior with
   implications.  The effects on security of not implementin
   SHOULD, or doing something the specification says MUST NO
   NOT be done may be very subtle. Document authors should t
   to elaborate the security implications of not following
   recommendations or requirements as most implementors will
   had the benefit of the experience and discussion that pro
   specification.

8. Acknowledgments

   The definitions of these terms are an amalgam of definiti
   from a number of RFCs.  In addition, suggestions have bee
   incorporated from a number of people including Robert Ull
   Narten, Neal McBurnett, and Robert Elz.
```

Tell Them What You've Told Them...

- **Requirements Documents** are **important** – e.g., for
  - negotiation, design & implementation, documentation, testing, delivery, re-use, re-implementation.
- A **Requirements Specification** should be
  - correct, complete, relevant, consistent, neutral, traceable, objective.

  Note: vague vs. abstract.
- **Requirements Representations** should be
  - easily understandable, precise, easily maintainable, easily usable
- **Distinguish**
  - hard / soft,
  - functional / non-functional,
  - open / tacit.
- It is the task of the **analyst** to **elicit** requirements.
- Natural language is inherently imprecise, counter-measures:
  - natural language patterns.
- Do not underestimate the value of a good **dictionary**.

References

References

Arenis, S. F., Westphal, B., Dietsch, D., Muñiz, M., and Andisha, A. S. (2014). The wireless fire alarm system: Ensuring conformance to industrial standards through formal verification. In Jones, C. B., Pihlajasaari, P., and Sun, J., editors, *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, volume 8442 of *LNCS*, pages 658–672. Springer.

Brooks, F. P. (1995). *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*. Addison-Wesley.

Gacitua, R., Ma, L., Nuseibeh, B., Piwek, P., de Roeck, A., Rouncefield, M., Sawyer, P., Willis, A., and Yang, H. (2009). Making tacit requirements explicit. talk.

IEEE (1990). *IEEE Standard Glossary of Software Engineering Terminology*. Std 610.12-1990.

IEEE (1998). *IEEE Recommended Practice for Software Requirements Specifications*. Std 830-1998.

Ludewig, J. and Lichter, H. (2013). *Software Engineering*. dpunkt.verlag, 3. edition.

Rupp, C. and die SOPHISTen (2009). *Requirements-Engineering und -Management*. Hanser, 5th edition.