Softwaretechnik / Software-Engineering

Lecture 5: Requirements Engineering

2019-05-13

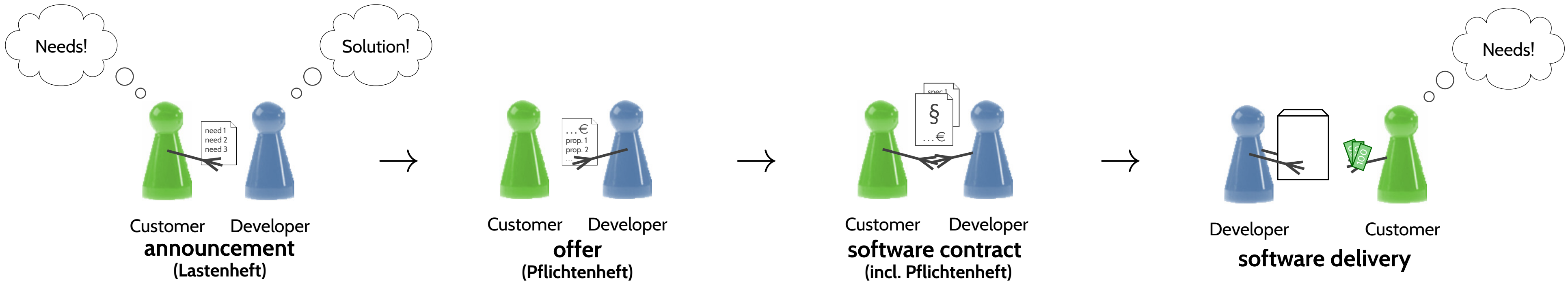
Prof. Dr. Andreas Podelski, **Dr. Bernd Westphal**

Albert-Ludwigs-Universität Freiburg, Germany

You Are Here.

	-	22.4., Mon
Introduction	L 1:	25.4., Thu
Metrics, Costs,	L 2:	29.4., Mon
Development	L 3:	2.5., Thu
Process	L 4:	6.5., Mon
	T 1:	9.5., Thu
	L 5:	13.5., Mon
Requirements	L 6:	16.5., Thu
Engineering	L 7:	20.5., Mon
	T 2:	23.5., Thu
	L 8:	27.5., Mon
	-	30.5., Thu
	L 9:	3.6., Mon
	T 3:	6.6., Thu
	-	10.6., Mon
	-	13.6., Thu
Arch. & Design,	L10:	17.6., Mon
	-	20.6., Thu
Software-	L 11:	24.6., Mon
	T 4:	27.6., Thu
Modelling,	L12:	1.7., Mon
Patterns	L13:	4.6., Thu
QA	L14:	8.7., Mon
	T 5:	11.7., Thu
(Testing, Formal	L15:	15.7., Mon
Verification)	L16:	18.7., Thu
Wrap-Up	L17:	22.7., Mon
	T 6:	25.7., Thu

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)

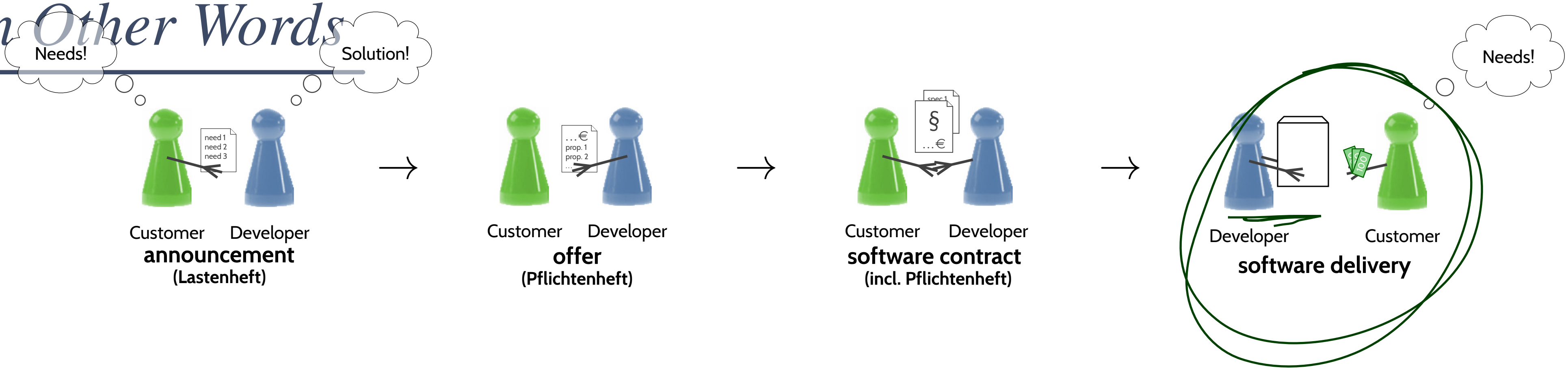
requirements analysis –

(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.

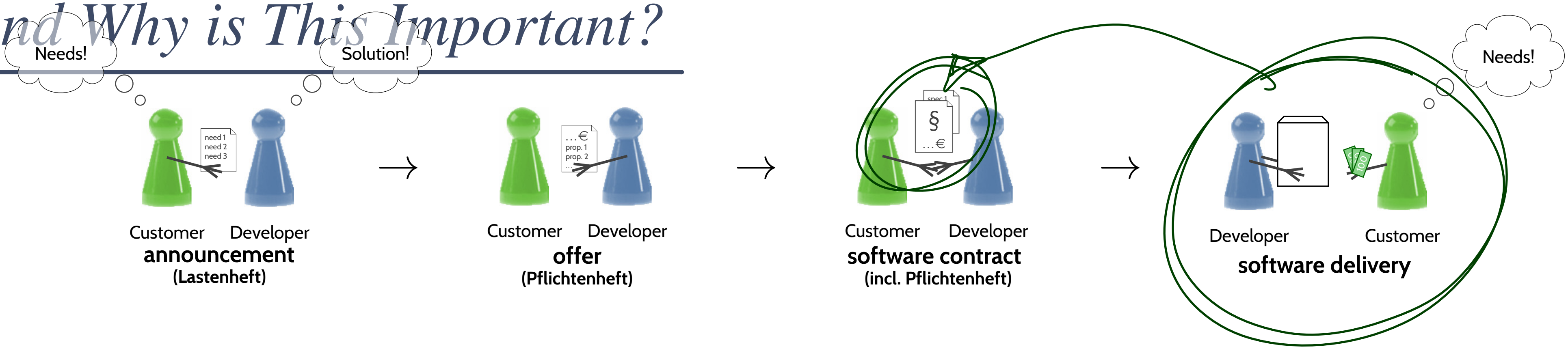
IEEE 610.12 (1990)

In Other Words



- A **requirements specification**,
- i.e., a set of requirements,
- is supposed to **partition**
- the set of **possible systems**
- into **acceptable** and **non-acceptable** (or correct and incorrect) systems.

And Why is This Important?



		customer's view: product is inacceptable / incorrect	customer's view: product is acceptable / correct
developer's view:	acceptable / correct	false negative	true positive
developer's view:	inacceptable / incorrect	true negative	false positive

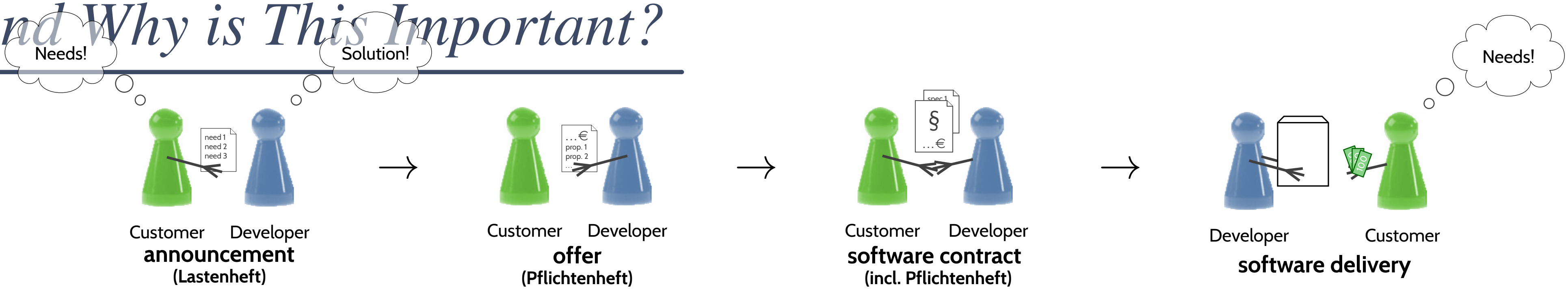
- Customer **accepts** product: Full payment from customer due, developer happy.

  (Unfortunate: customer may still be unhappy with the delivered product!)

- Customer **does not accept** product: No full payment, developer unhappy.

  → usually both parties unhappy, everybody should want to avoid this situation.

And Why is This Important?



		customer's view: product is inacceptable / incorrect	customer's view: product is acceptable / correct
developer's view:	acceptable / correct	false negative	true positive
developer's view:	inacceptable / incorrect	true negative	false positive ✗

Judge
may be consulted to decide the dispute, will consider specification;
may follow developer or customer or …

- Customer **accepts** product: Full payment from customer due, developer happy.

  (Unfortunate: customer may still be unhappy with the delivered product!)
- Customer **does not accept** product: No full payment, developer unhappy.

  → usually both parties unhappy, everybody should want to avoid this situation.

Software, formally

Definition. **Software** is a finite description S of a (possibly infinite) set $[\![S]\!]$ of (finite or infinite) computation paths of the form

$$\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots$$

where

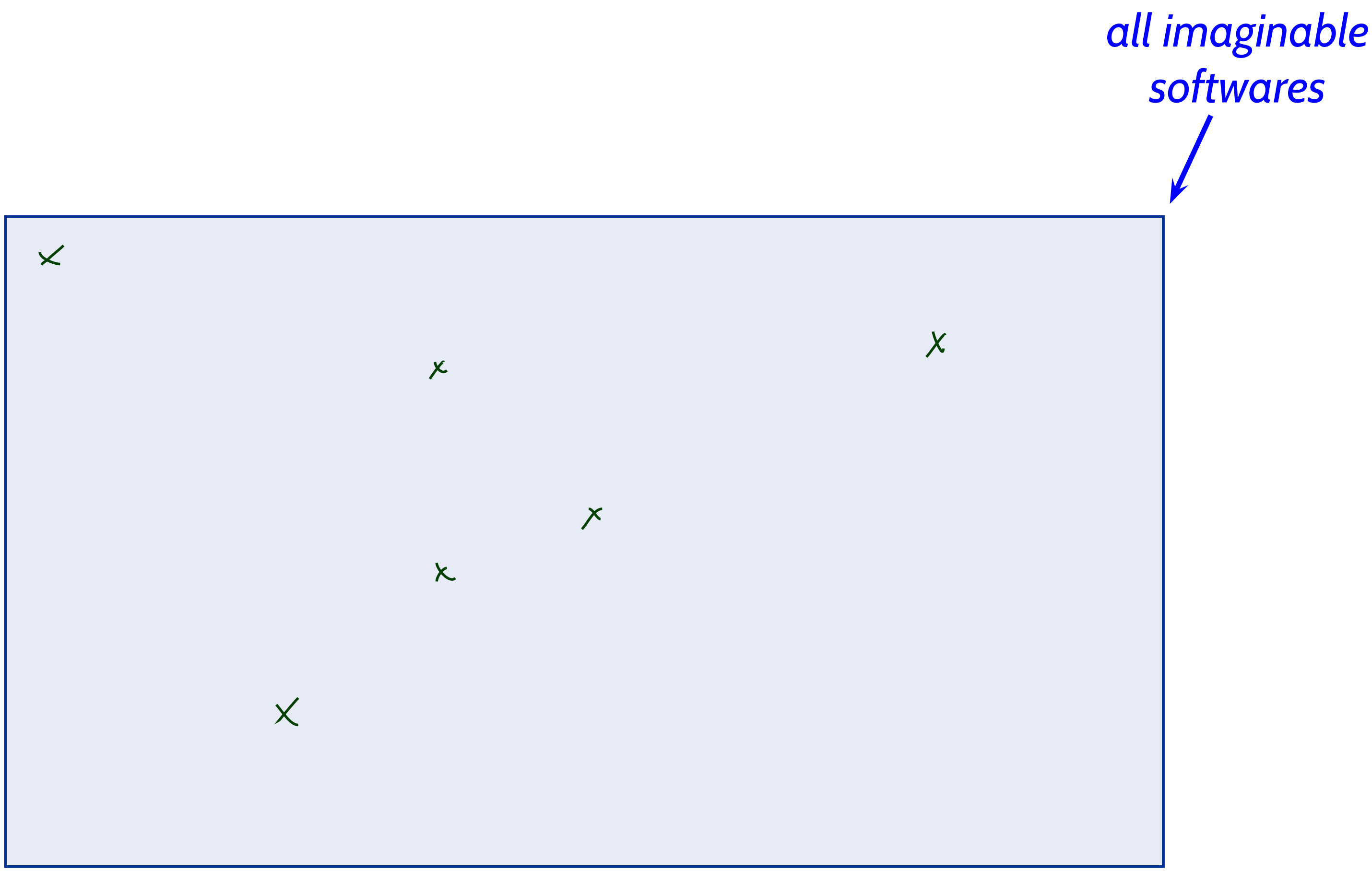
- $\sigma_i \in \Sigma$, $i \in \mathbb{N}_0$, is called **state** (or **configuration**), and
- $\alpha_i \in A$, $i \in \mathbb{N}_0$, is called **action** (or **event**).

The (possibly partial) function $[\![\,\cdot\,]\!] : S \mapsto [\![S]\!]$ is called **interpretation** of S.

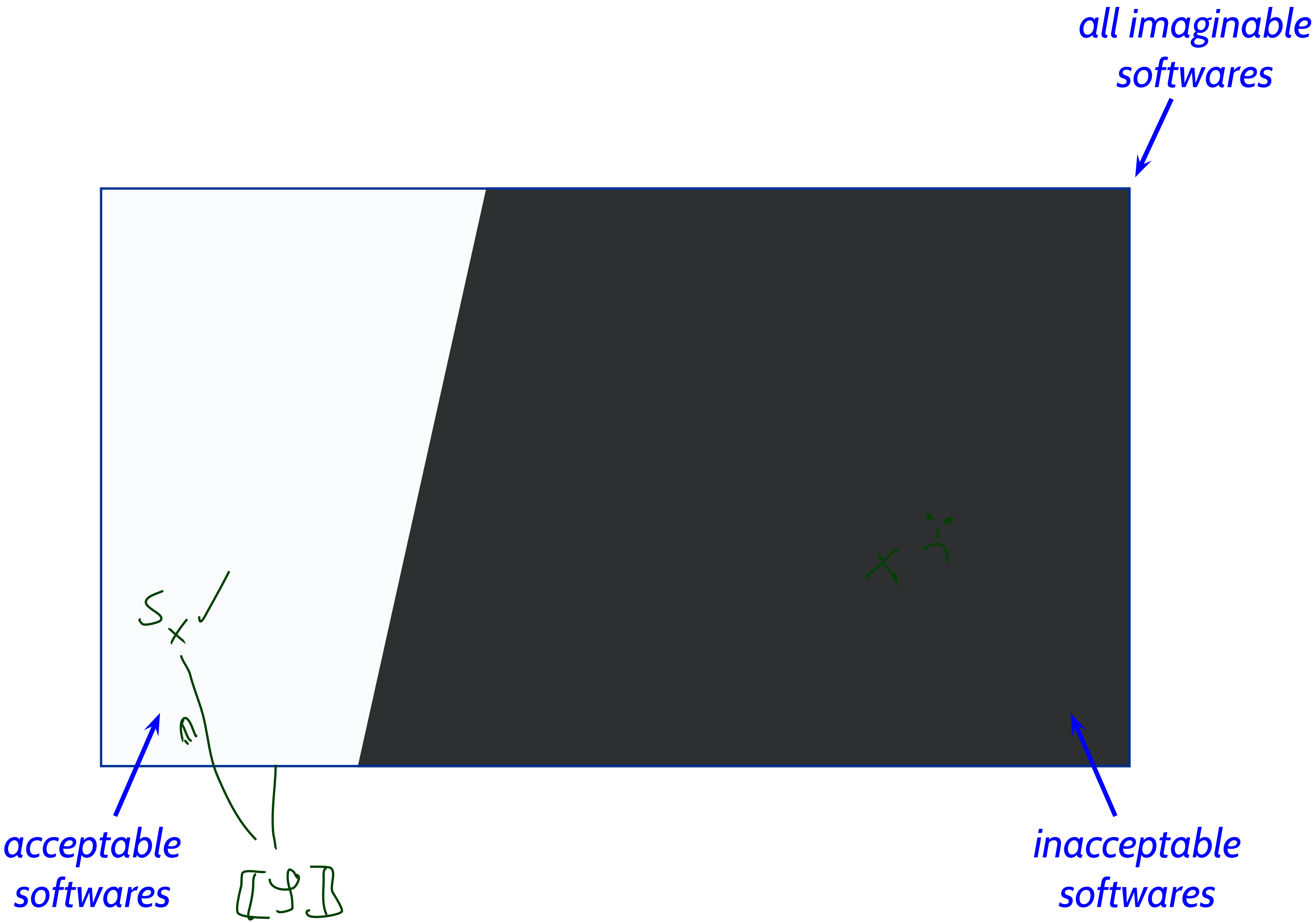
Examples:

- '**Hallo**' (from Lect. 2): Can be seen as having one computation path.
- a **Quicksort** implementation: Can be seen as having as many computation paths as possible inputs.
- **Pedestrations Crossing controller**: Usually has infinitely many computation paths (each sequence of pedestrians pressing button at particular times defines a different computation path).
- etc.
- **Note**: one software S may have different interpretations, ranging from 'only final result' (coarse; if well-defined) to 'register transfer level' (fine), with or without time-stamps, etc..

Software Specification: An Ideal Partitioning

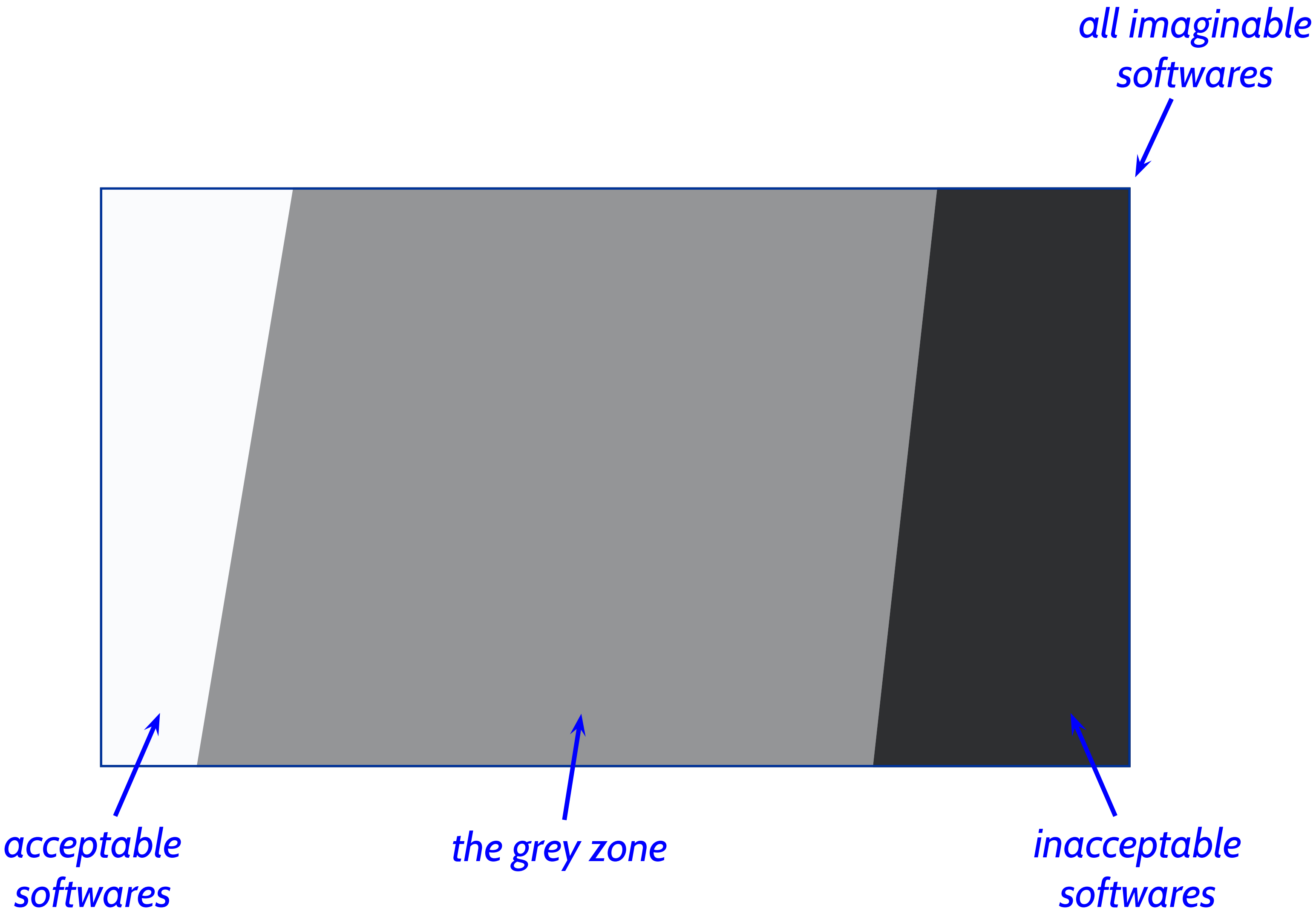


Software Specification: An Ideal Partitioning

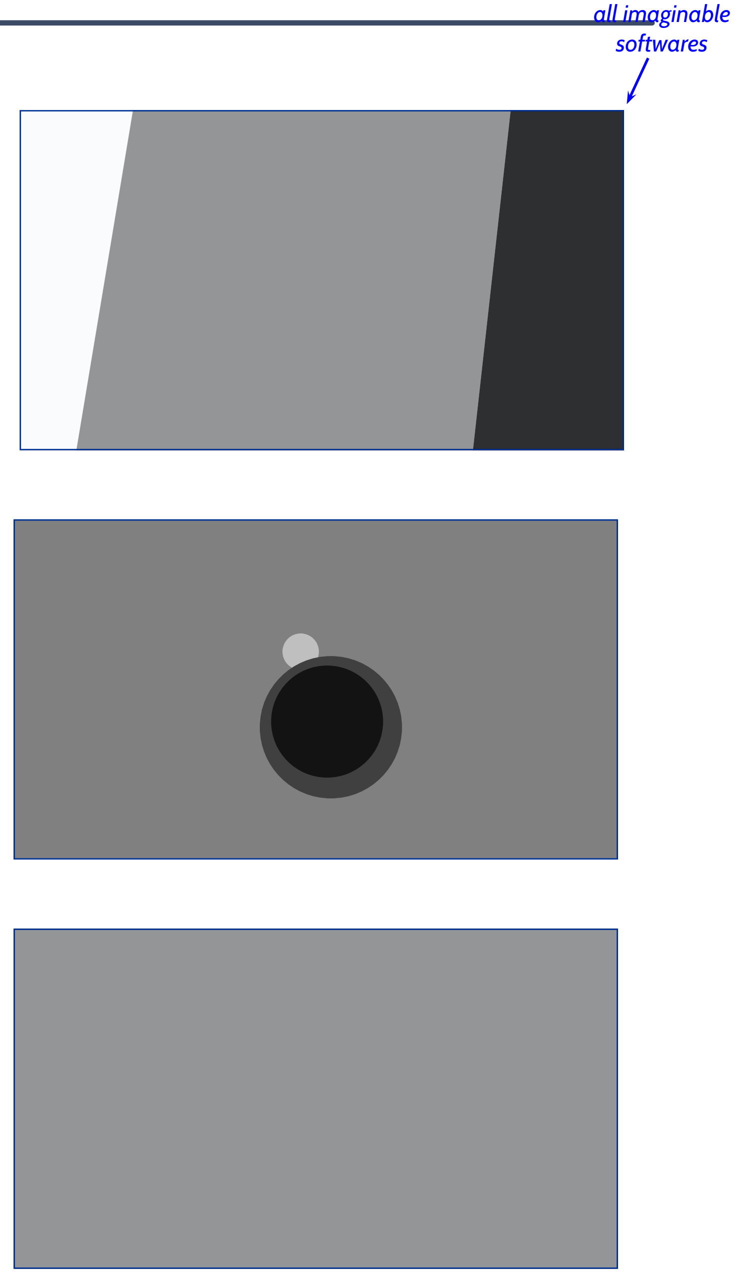


Software Specification: Perceived Practice

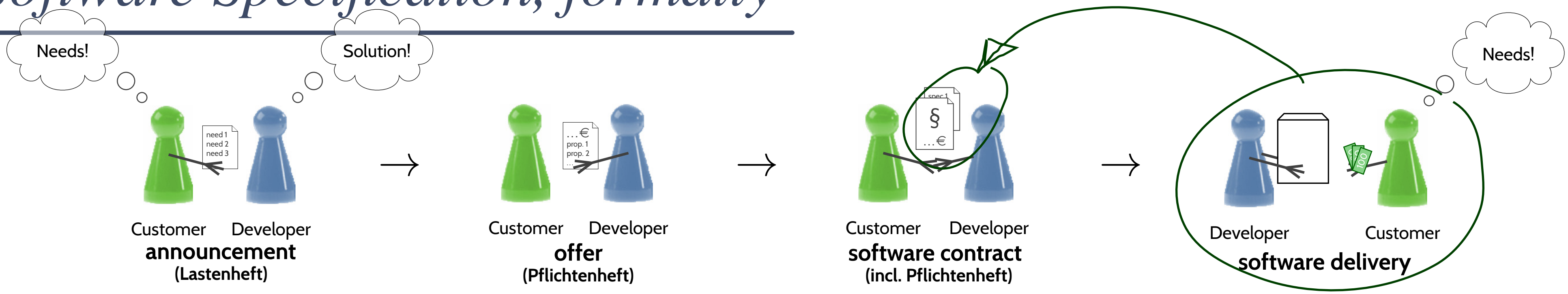
Software Specification: Perceived Practice



Software Specification: Perceived Practice



Software Specification, formally



Definition. A **software specification** is a finite description $\mathscr{S}$ of a (possibly infinite) set $[\![\mathscr{S}]\!]$ of softwares, i.e.

$$[\![\mathscr{S}]\!] = \{(S_1, [\![\,\cdot\,]\!]_1), (S_2, [\![\,\cdot\,]\!]_2), \dots\}.$$

The (possibly partial) function $[\![\,\cdot\,]\!] : \mathscr{S} \mapsto [\![\mathscr{S}]\!]$ is called **interpretation** of $\mathscr{S}$.

Definition. Software $(S, [\![\,\cdot\,]\!])$ **satisfies** software specification $\mathscr{S}$, denoted by $S \models \mathscr{S}$, if and only if

$$(S, [\![\,\cdot\,]\!]) \in [\![\mathscr{S}]\!].$$

Risks Implied by Bad Requirements Specifications

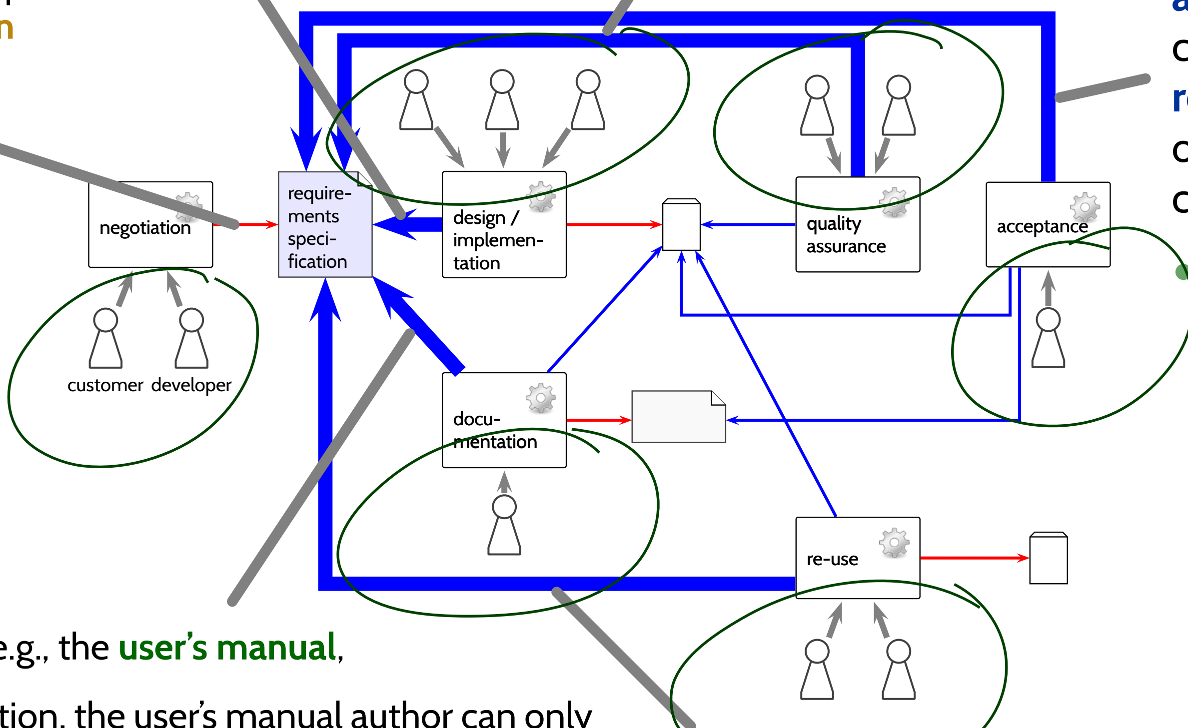
design and **implementation**,

- without specification, programmers may just "ask around" when in doubt, possibly yielding different interpretations → **difficult integration**

preparation of **tests**,

- without a description of allowed outcomes, tests are randomly searching for generic errors (like crashes) → **systematic testing hardly possible**

negotiation (with customer, marketing department, or . . .)



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**

documentation, e.g., the **user's manual**,

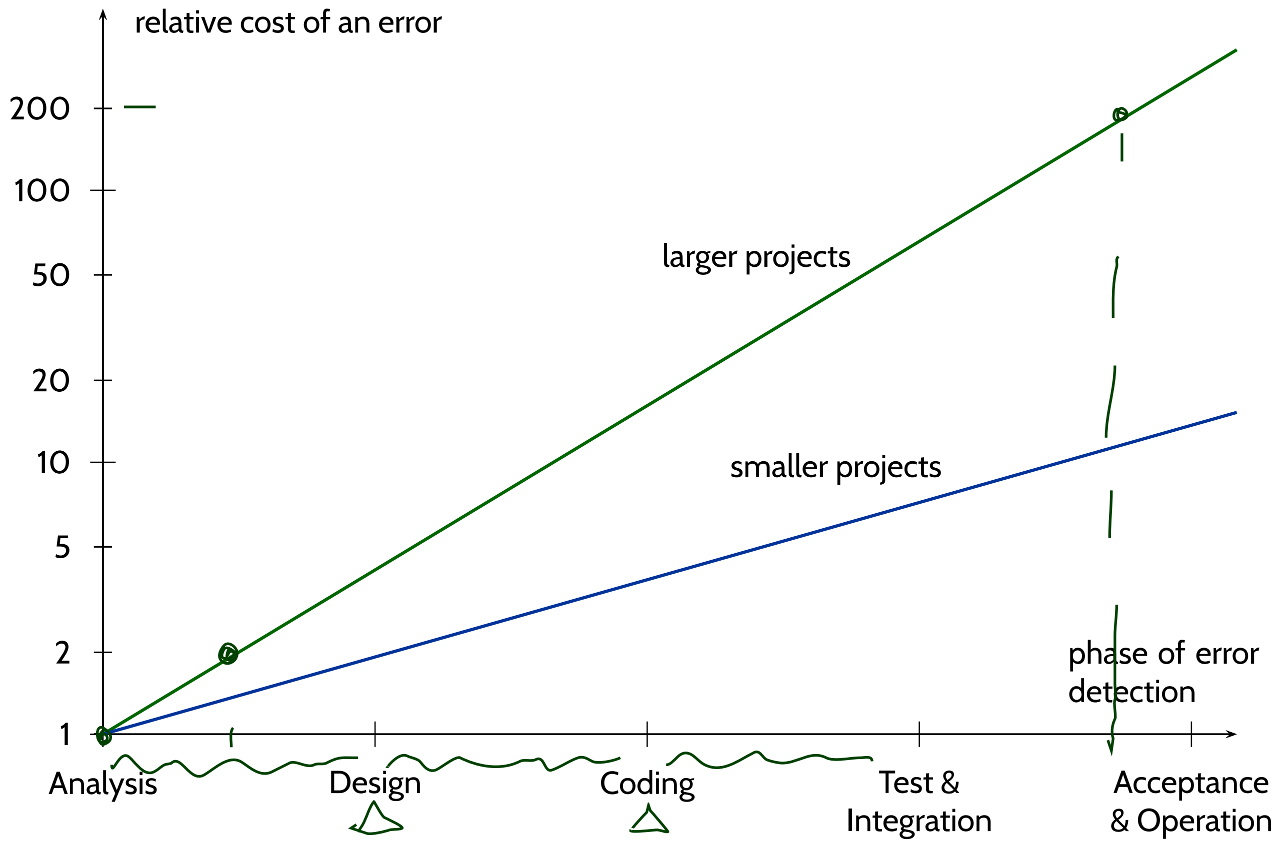
- without specification, the user's manual author can only describe what the system **does**, not what it should do (**"every observation is a feature"**)

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**

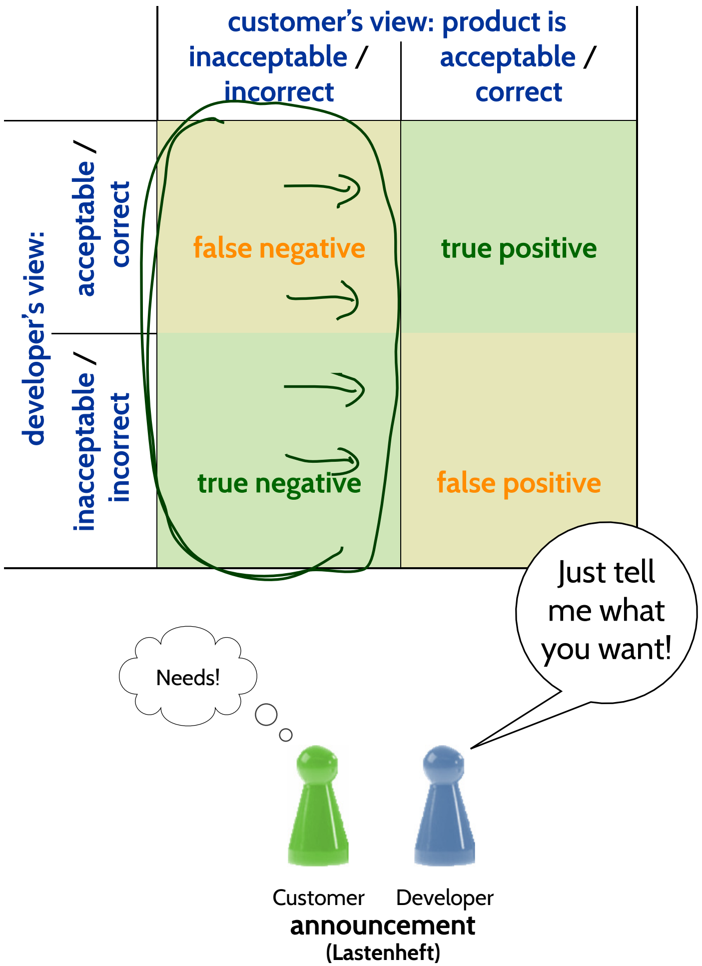
Discovering Fundamental Errors Late Can Be Expensive



Relative error costs over latency according to investigations at IBM, etc.

By (Boehm, 1979); Visualisation: Ludewig and Lichter (2013).

Getting Requirements Right



$\rightarrow$ does not work in general.

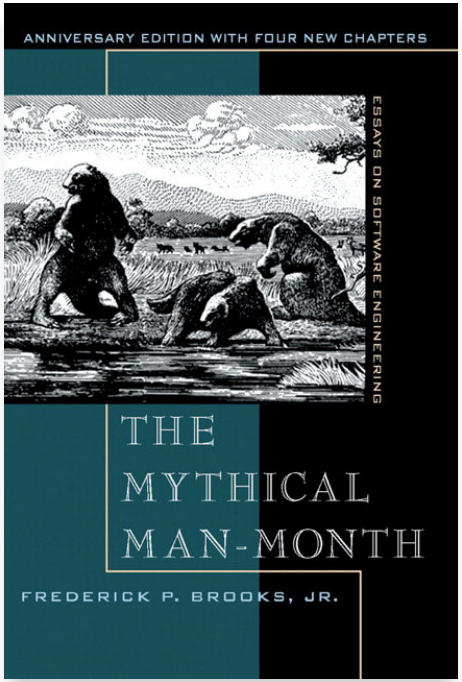
- Analogy: Most people **couldn't even specify a bicycle** – they feel that they can, because bicycle manufacturers do the work for us. With software, we are not yet there.

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)

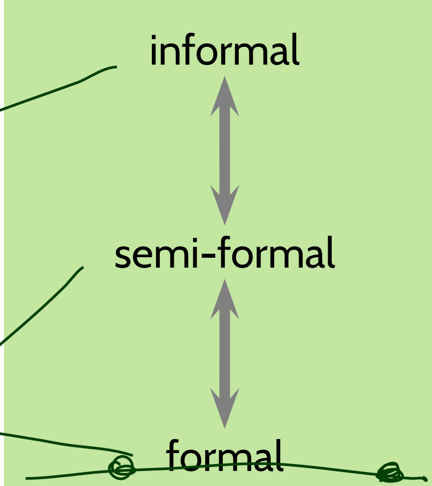


Topic Area Requirements Engineering: Content

VL 5	• **Introduction**
	• **Requirements Specification**
	– Desired Properties
	– Kinds of Requirements
	– Analysis Techniques
⋮	• **Documents**
	– Dictionary, Specification
	• **Specification Languages**
	– Natural Language
VL 6	– Decision Tables
	– Syntax, Semantics
⋮	– Completeness, Consistency, …
VL 7	– Scenarios
⋮	– User Stories, Use Cases
VL 8	– Live Sequence Charts
	– Syntax, Semantics
⋮	• Definition: **Software** & **SW Specification**
VL 9	• **Wrap-Up**
⋮	

Vocabulary

Techniques



Content

- **Introduction**
  - Vocabulary: Requirements (Analysis)
  - Importance of Requirements Specifications
- **Requirements Specification**
  - Requirements Analysis
  - Desired Properties
  - Kinds of Requirements
  - Analysis Techniques
- **Documents**
  - Dictionary
  - Specification
- **Requirements Specification Languages**
  - Natural Language

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.

  $\rightarrow$ is is **difficult** (if at all possible) to **be sure of correctness** and **completeness**.

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 (→ 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.

  → provide redundant explanations.

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

  → **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 too 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?").

Content

Kinds of Requirements

Kinds of Requirements: Functional and Non-Functional

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

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

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,
  - $o_1, o_2, \ldots$: 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.

Content

Requirements Analysis Techniques

Requirements Engineers See the World Differently

- The human brain is great at **seeing information** (even if there isn't so much);
- **Requirements Engineering** is about **seeing the absence of information**.

Example: Wireless Fire Alarm System

The loss of the ability of the system to transmit a signal from a component to the central unit is

- detected in less than 300 seconds and
- displayed at the central unit within 100 seconds thereafter.

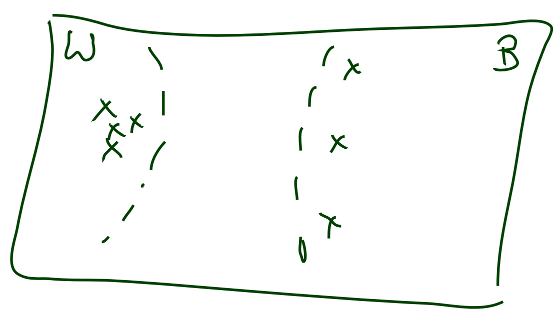
Requirements Elicitation

- **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** ('Herauskitzeln') the requirements.



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** ('Herauskitzeln') the requirements.

- 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.
  - Sort/assess resulting "**raw material**" in half-/full-day workshops in 6-10 people team.

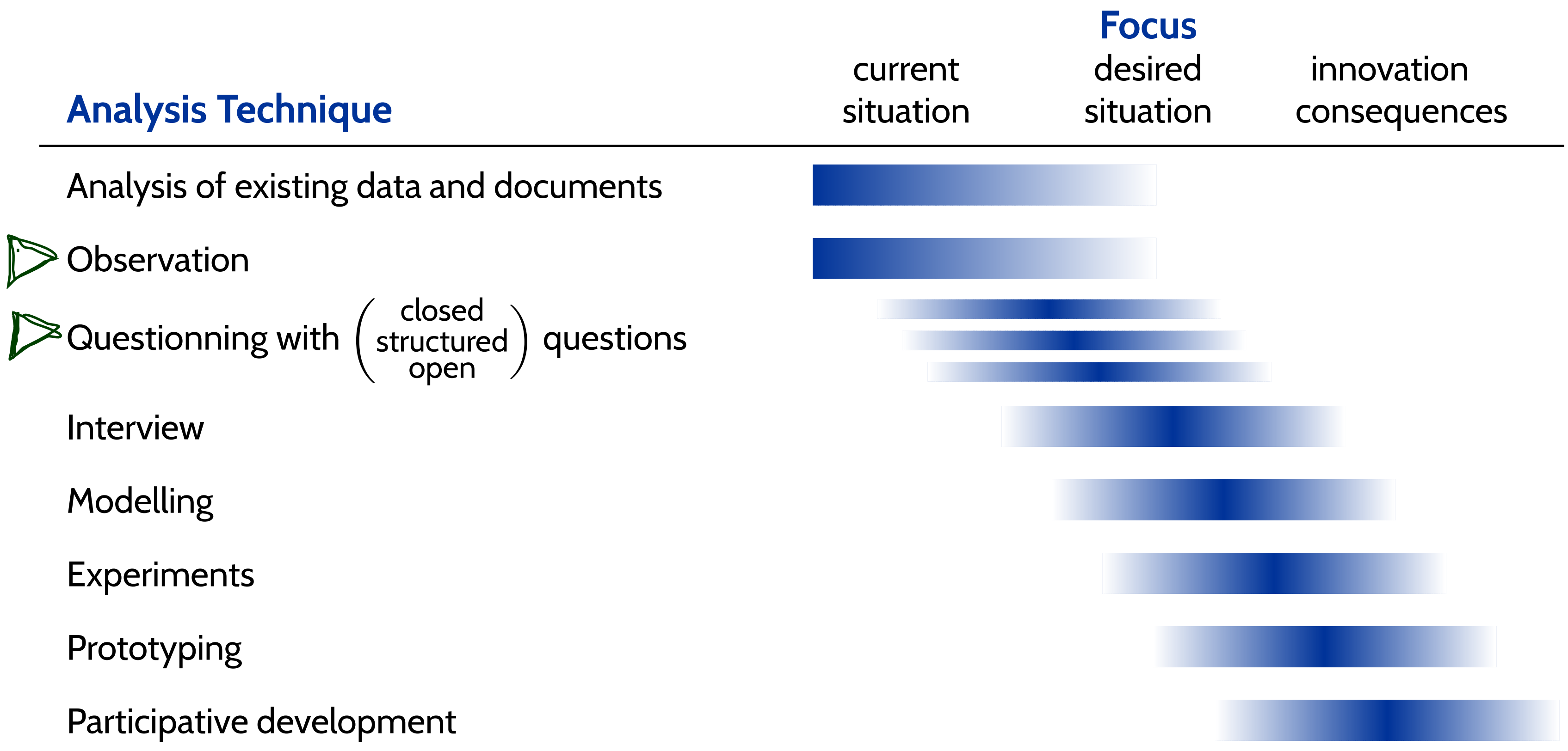
    Search for, e.g., **contradictions** between customer wishes, and for **priorisation**.

    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**.

(A Selection of) Analysis Techniques



(Ludewig and Lichter, 2013)

Content

Requirements Documents

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)

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.

Boehm, B. W. (1979). Guidelines for verifying and validating software requirements and design specifications. In *EURO IFIP 79*, pages 711–719. Elsevier North-Holland.

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.