Softwaretechnik / Software-Engineering

Lecture 05: Requirements Engineering

2015-05-11

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

Albert-Ludwigs-Universität Freiburg, Germany

You Are Here

Course Content (Original Plan)

Introduction	L 1:	20.4., Mo
	T 1:	23.4., Do
Development Process, Metrics	L 2:	27.4., Mo
	L 3:	30.4., Do
	L 4:	4.5., Mo
	T 2:	7.5., Do
Requirements Engineering	L 5:	11.5., Mo
	-	14.5., Do
	L 6:	18.5., Mo
	L 7:	21.5., Do
	-	25.5., Mo
	-	28.5., Do
	T 3:	1.6., Mo
Design Modelling & Analysis	-	4.6., Do
	L 8:	8.6., Mo
	L 9:	11.6., Do
	L 10:	15.6., Mo
	T 4:	18.6., Do
Implementation, Testing	L 11:	22.6., Mo
	L 12:	25.6., Do
	L 13:	29.6., Mo
	T 5:	2.7., Do
Formal Verification	L 14:	6.7., Mo
	L 15:	9.7., Do
	L 16:	13.7., Mo
	T 6:	16.7., Do
The Rest	L 17:	20.7., Mo
	L 18:	23.7., Do

Contents & Goals

Last Lecture:

- process models: V-Modell XT, agile (XP, Scrum); process metrics: CMMI, SPICE

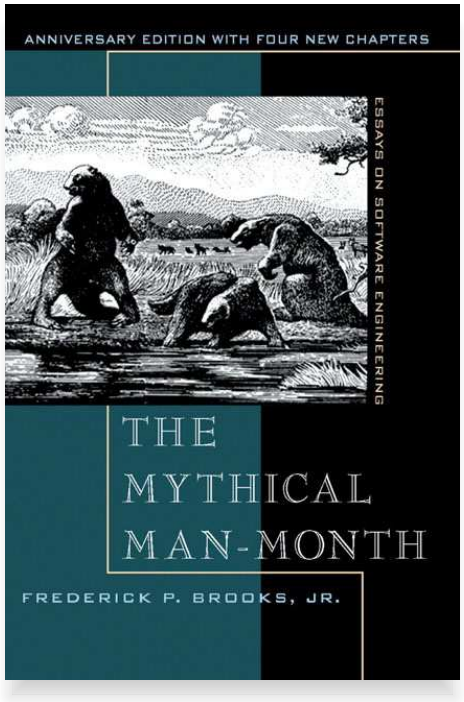
This Lecture:

- **Educational Objectives:** Capabilities for following tasks/questions.
  - What is requirements engineering (RE)?
  - Why is it important, why is it hard?
  - What are the two (three) most relevant artefacts produced by RE activities?
  - What is a dictionary?
  - What are desired properties of requirements specification (documents)?
  - What are hard/soft/open/tacit/functional/non-functional requirements?
  - What is requirements elicitation?
  - Which analysis technique would you recommend in which situation?
- **Content:**
  - motivation and vocabulary of requirements engineering,
  - the documents of requirements analysis, and desired properties of RE,
  - guidelines for requirements specification using natural language

Customer Developer

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)



Requirements and Requirements Analysis

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)

Requirements and Requirements Analysis

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)

Requirements and Requirements Analysis

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).

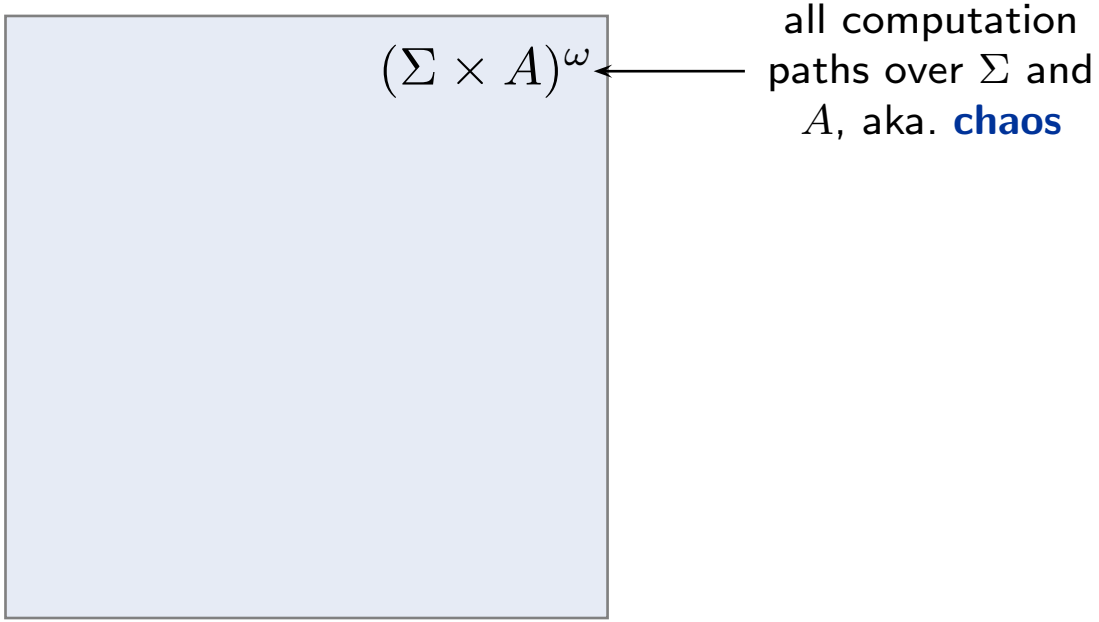
IEEE 610.12 (1990)

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.

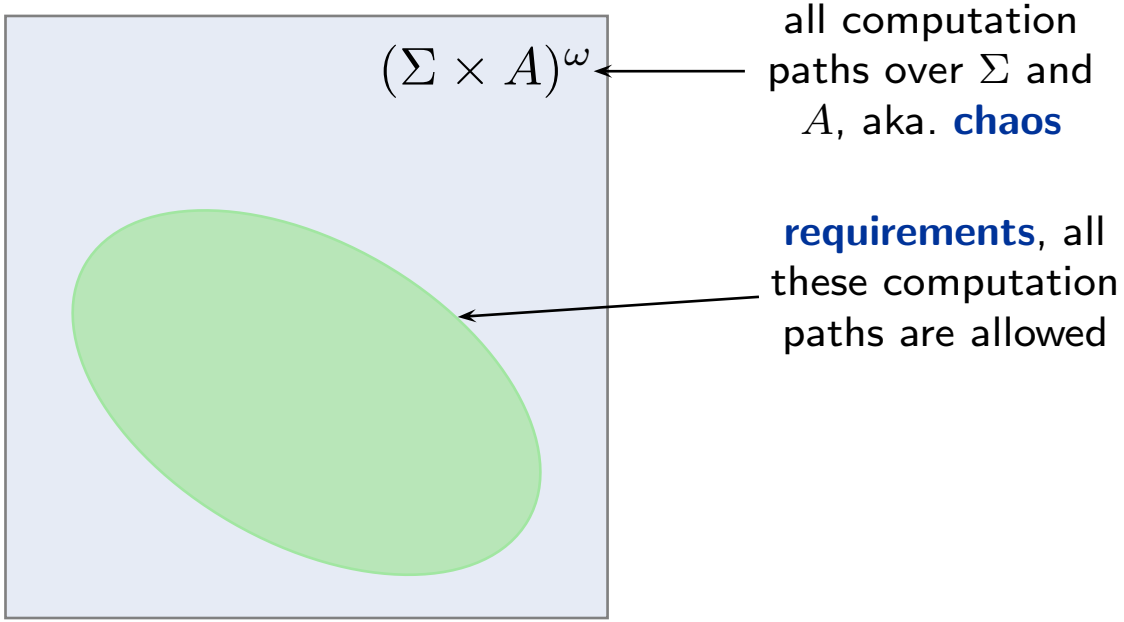
IEEE 610.12 (1990)

The Requirements Engineering Problem

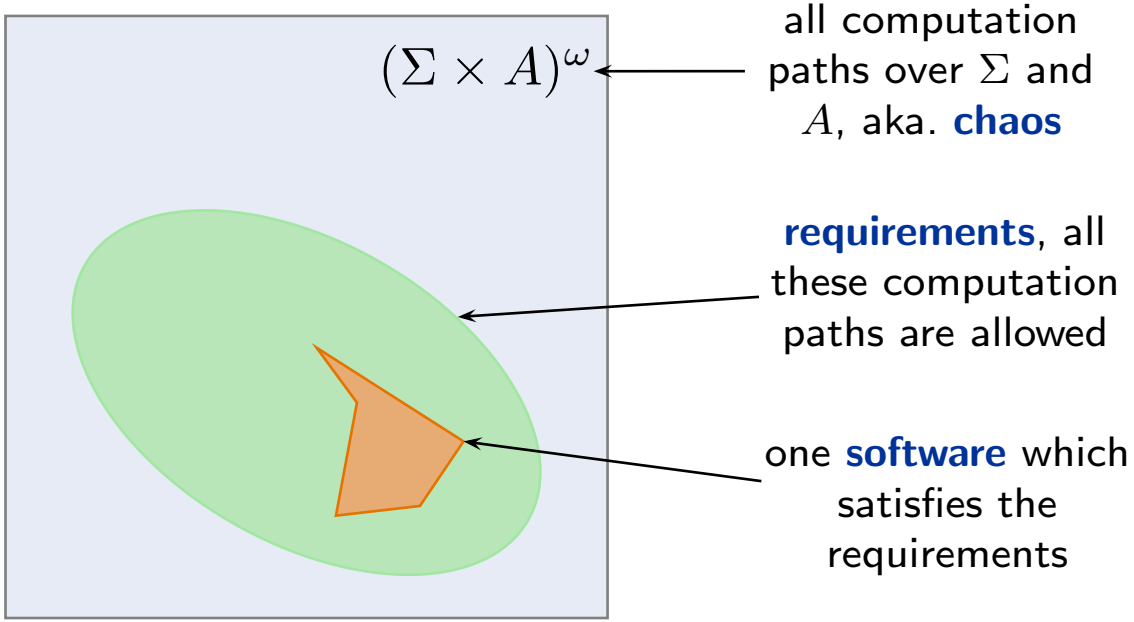
The Requirements Engineering Problem



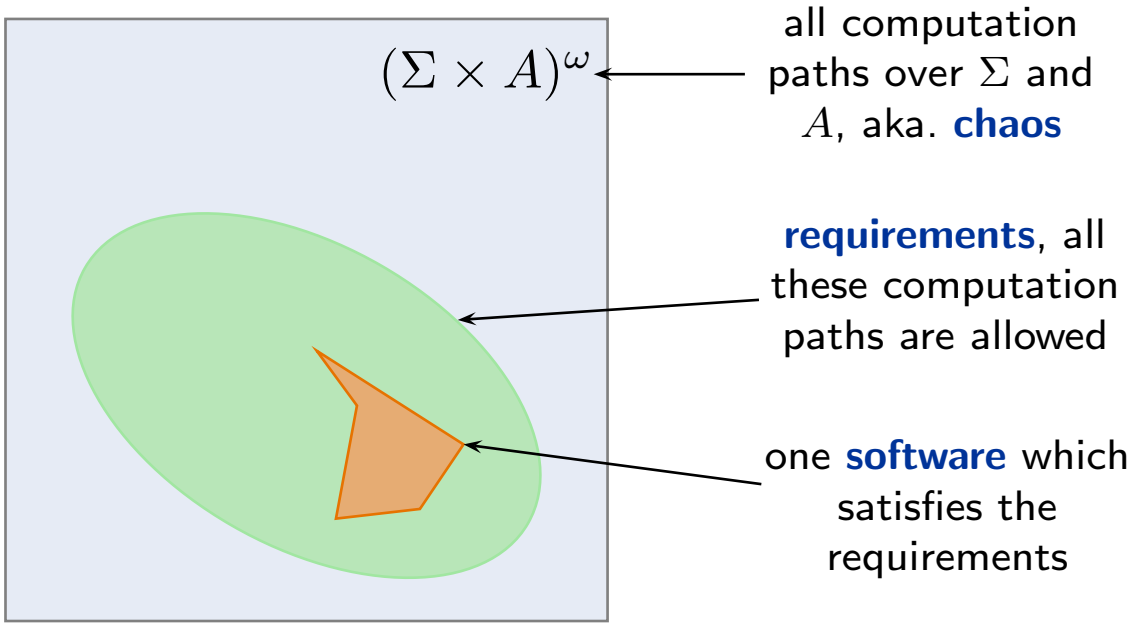
The Requirements Engineering Problem



The Requirements Engineering Problem



The Requirements Engineering Problem



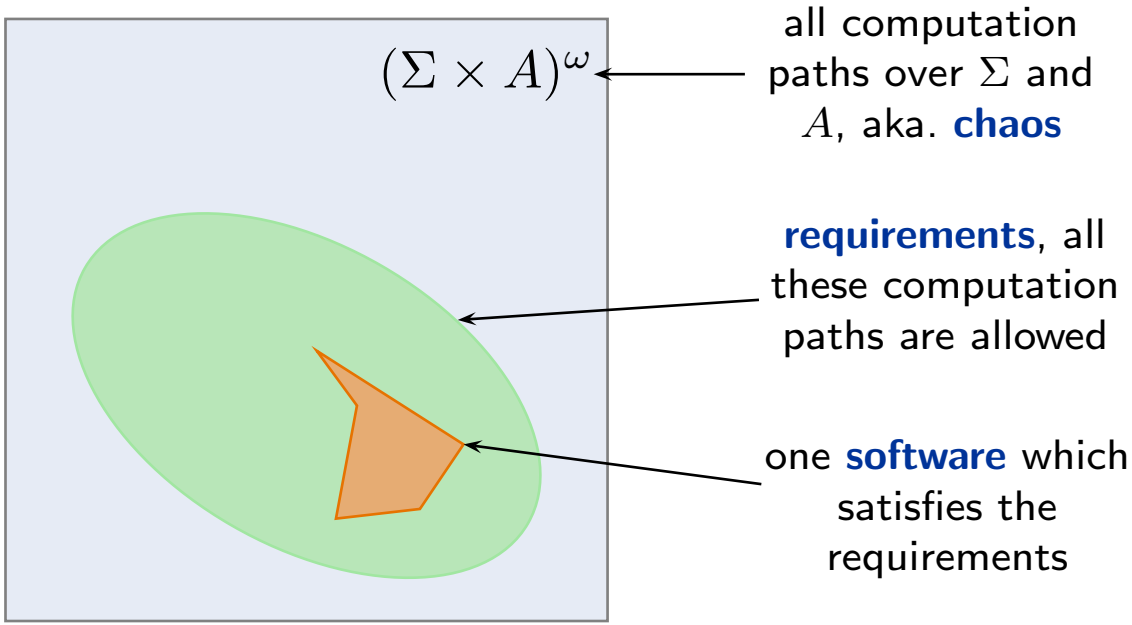
- **Requirements engineering**:

  Describe/specify the set of the **allowed** computation paths.

- **Software development**:

  Create one software S whose computation paths $[\![S]\!]$ are all allowed.

The Requirements Engineering Problem



- **Requirements engineering**:

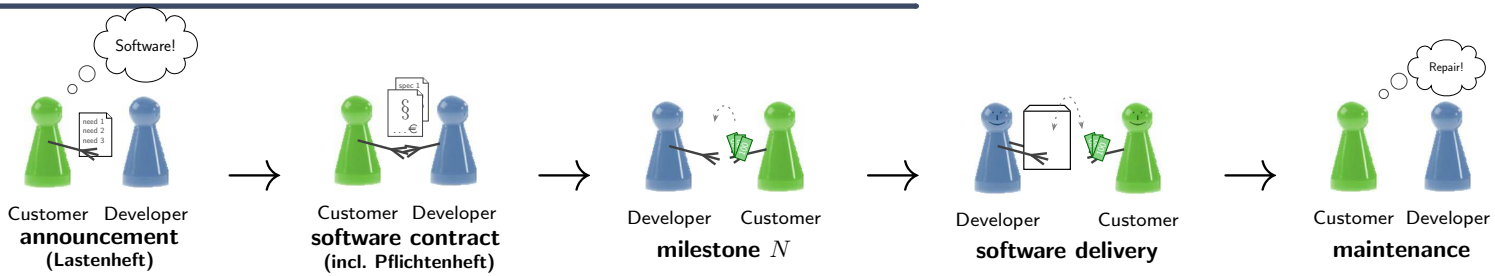
  Describe/specify the set of the **allowed** computation paths.

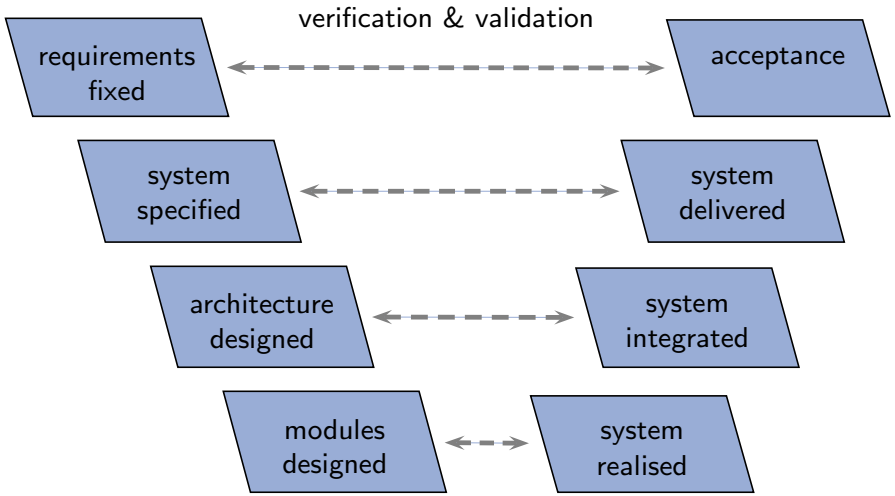
- **Software development**:

  Create one software S whose computation paths $[\![S]\!]$ are all allowed.

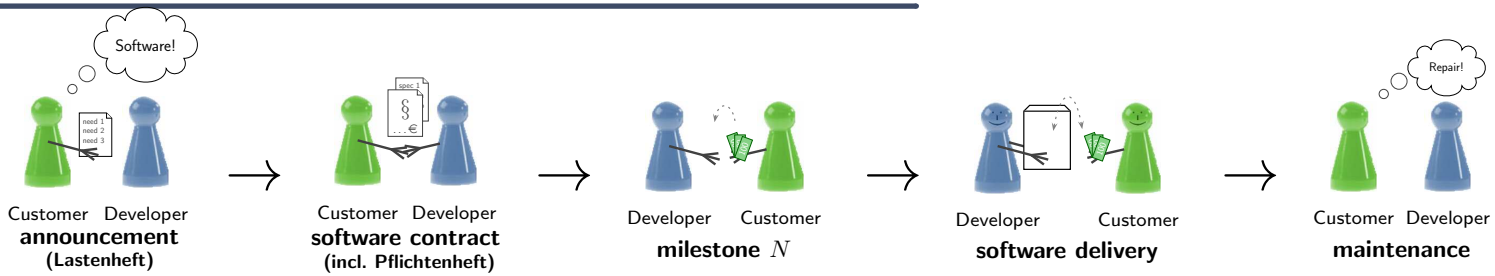
  - **Note**: different programs in different programming languages may also describe $[\![S]\!]$.
  - **Often allowed**: any **refinement** of  ($\rightarrow$ later; e.g. allow intermediate transitions).

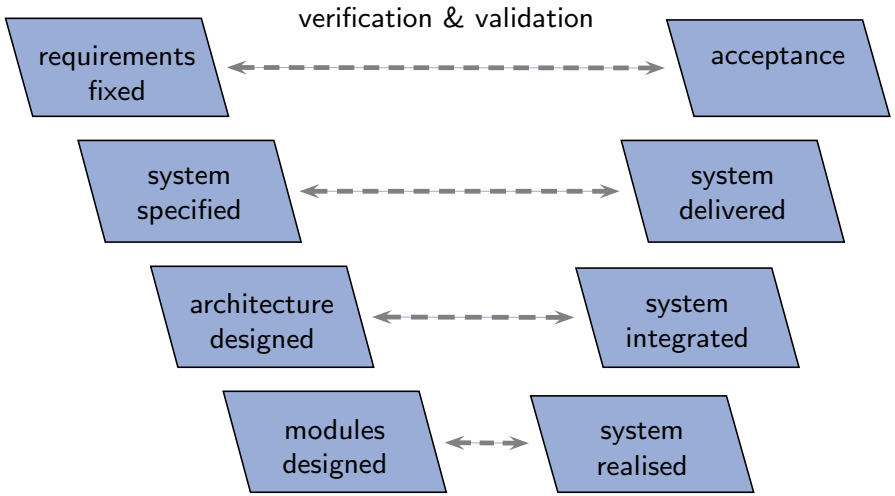
So What is So Important About That?





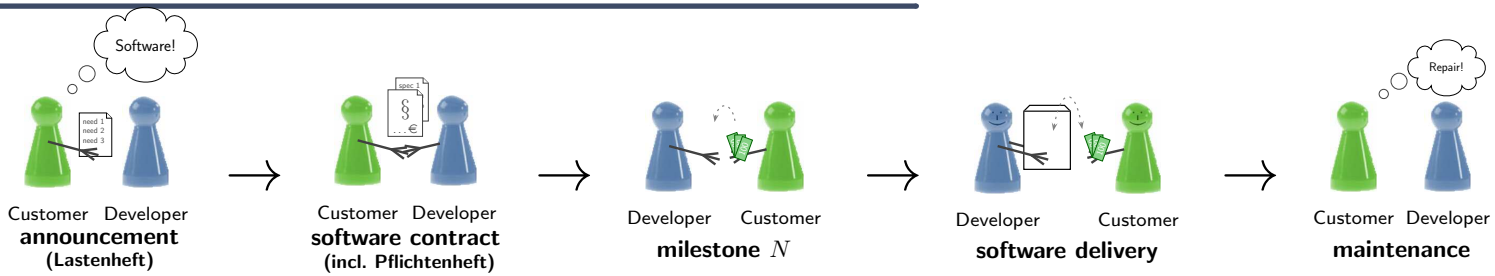
So What is So Important About That?

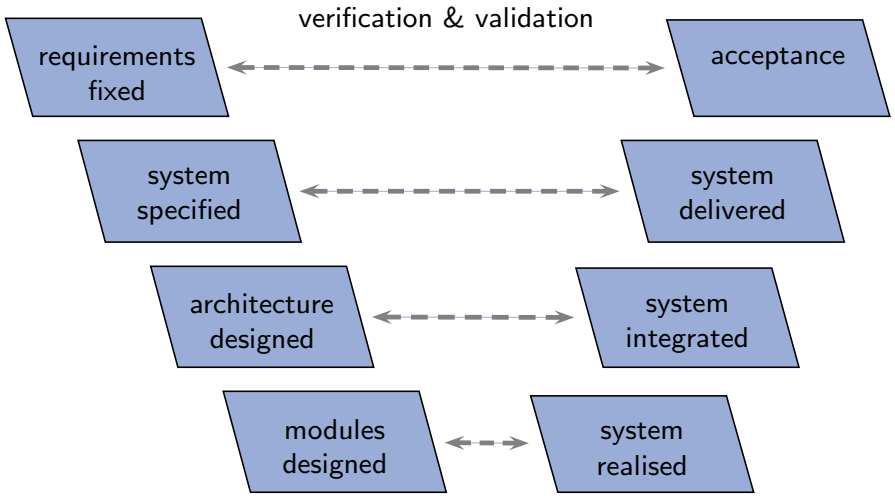




- the **documentation** of the requirements defines acceptance criteria, thus will be considered in any dispute at software delivery time!
  (plus, speaking of documentation, mind the → **bus-factor**)

So What is So Important About That?





- the **documentation** of the requirements defines acceptance criteria, thus will be considered in any dispute at software delivery time! (plus, speaking of documentation, mind the → **bus-factor**)
- actively discussed in industry these days: **traceability**

Purposes of the Requirements Specification

Purposes of the Requirements Specification

- **coordination** with the customer (or the marketing department, or . . .)
  - not properly clarified/specified requirements are hard to satisfy — mismatches with customer's needs turn out in operation the latest → **additional effort**

Purposes of the Requirements Specification

- **coordination** with the customer (or the marketing department, or . . .)
  - not properly clarified/specified requirements are hard to satisfy — mismatches with customer's needs turn out in operation the latest → **additional effort**
- **design** and **implementation**,
  - programmers may use different interpretations of unclear requirements → **difficult integration**

Purposes of the Requirements Specification

- **coordination** with the customer (or the marketing department, or . . .)
  - not properly clarified/specified requirements are hard to satisfy — mismatches with customer's needs turn out in operation the latest → **additional effort**
- **design** and **implementation**,
  - programmers may use different interpretations of unclear requirements → **difficult integration**
- the user's **manual**,
  - if the user's manual author is not developer, he/she can only describe what the system does, not what it should do (**"no more bugs, every observation is a feature"**)

Purposes of the Requirements Specification

- **coordination** with the customer (or the marketing department, or . . .)
  - not properly clarified/specified requirements are hard to satisfy — mismatches with customer's needs turn out in operation the latest → **additional effort**
- **design** and **implementation**,
  - programmers may use different interpretations of unclear requirements → **difficult integration**
- the user's **manual**,
  - if the user's manual author is not developer, he/she can only describe what the system does, not what it should do (**"no more bugs, 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 impossible**

Purposes of the Requirements Specification

- **coordination** with the customer (or the marketing department, or . . .)
  - not properly clarified/specified requirements are hard to satisfy — mismatches with customer's needs turn out in operation the latest → **additional effort**
- **design** and **implementation**,
  - programmers may use different interpretations of unclear requirements → **difficult integration**
- the user's **manual**,
  - if the user's manual author is not developer, he/she can only describe what the system does, not what it should do (**"no more bugs, 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 impossible**
- **acceptance** by customer, **resolving** later objections or regress claims,
  - at delivery time unclear whether behaviour is an error (developer needs to fix) or correct (customer needs to accept and pay) → **nasty disputes**, **additional effort**

Purposes of the Requirements Specification

- **coordination** with the customer (or the marketing department, or . . .)
  - not properly clarified/specified requirements are hard to satisfy — mismatches with customer's needs turn out in operation the latest → **additional effort**
- **design** and **implementation**,
  - programmers may use different interpretations of unclear requirements → **difficult integration**
- the user's **manual**,
  - if the user's manual author is not developer, he/she can only describe what the system does, not what it should do (**"no more bugs, 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 impossible**
- **acceptance** by customer, **resolving** later objections or regress claims,
  - at delivery time unclear whether behaviour is an error (developer needs to fix) or correct (customer needs to accept and pay) → **nasty disputes**, **additional effort**
- **re-use**,
  - re-use based on re-reading the code → **risk of unexpected changes**

Purposes of the Requirements Specification

- **coordination** with the customer (or the marketing department, or . . .)
  - not properly clarified/specified requirements are hard to satisfy — mismatches with customer's needs turn out in operation the latest → **additional effort**
- **design** and **implementation**,
  - programmers may use different interpretations of unclear requirements → **difficult integration**
- the user's **manual**,
  - if the user's manual author is not developer, he/she can only describe what the system does, not what it should do (**"no more bugs, 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 impossible**
- **acceptance** by customer, **resolving** later objections or regress claims,
  - at delivery time unclear whether behaviour is an error (developer needs to fix) or correct (customer needs to accept and pay) → **nasty disputes**, **additional effort**
- **re-use**,
  - re-use 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**

And What's Hard About That?

Once Again: The Software Peoples' View on Requirements

Once Again: The Software Peoples' View on Requirements

Example: children's birthday present requirements (birthday on May, 27th).

- "Ich will'n **Pony**!" ("I want a **pony**!")
- **Common sense understanding**:

Once Again: The Software Peoples' View on Requirements

Example: children's birthday present requirements (birthday on May, 27th).

- "Ich will'n **Pony**!" ("I want a **pony**!")
- **Common sense understanding**:

http://pixabay.com/en/pony-horse-animal-ride-pasture-368975/ — CC0 Public Domain

✔

"Ford Mustang on Felixstowe beach" by Steve Arnold — CC BY 2.0

✘

That is: we're looking for **one** small **horse-like animal**; **we may** (guided by economic concerns, taste, etc.) **choose** exact breed, size, color, shape, gender, age (may not even be born today, only needs to be alive on birthday)...

Once Again: The Software Peoples' View on Requirements

Example: children's birthday present requirements (birthday on May, 27th).

- "Ich will'n **Pony**!" ("I want a **pony**!")
- **Common sense understanding**:

That is: we're looking for **one** small **horse-like animal**; **we may** (guided by economic concerns, taste, etc.) **choose** exact breed, size, color, shape, gender, age (may not even be born today, only needs to be alive on birthday)...

- **Software Engineering understanding**:

Once Again: The Software Peoples' View on Requirements

Example: children's birthday present requirements (birthday on May, 27th).

- "Ich will'n **Pony**!" ("I want a **pony**!")
- **Common sense understanding**:

http://pixabay.com/en/pony-horse-animal-ride-pasture-368975/ — CC0 Public Domain

"Ford Mustang on Felixstowe beach" by Steve Arnold — CC BY 2.0

That is: we're looking for **one** small **horse-like animal**; **we may** (guided by economic concerns, taste, etc.) **choose** exact breed, size, color, shape, gender, age (may not even be born today, only needs to be alive on birthday)...

- **Software Engineering understanding**:

  We may give **everything** as long as **there's a pony in it**:

  - a **herd of ponies**,
  - a **whole zoo** (if it has a pony),
  - ...

A Bit More Abstract: Software vs. 'Real World' Requirements

In other words:

- **common sense understanding**: choose from "$\emptyset \cup requirements$"
- **software engineering understanding**:

choose from "$chaos \cap requirements$"

A Bit More Abstract: Software vs. 'Real World' Requirements

In other words:

- **common sense understanding**: choose from "$\emptyset \cup requirements$"
- **software engineering understanding**:

  choose from "$chaos \cap requirements$"

- $\mathscr{S}$ = "x is always 0" is a (semi-informal) **software specification**.
  - It denotes **all imaginable softwares** with an x which is always 0:

$$[\![\mathscr{S}]\!] = \{S \mid \forall\, \sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots \in [\![S]\!]\ \forall\, i \in \mathbb{N}_0 \bullet \sigma_i \models x = 0\}$$

A Bit More Abstract: Software vs. 'Real World' Requirements

In other words:

- **common sense understanding**: choose from "$\emptyset \cup requirements$"
- **software engineering understanding**:

  choose from "$chaos \cap requirements$"

- $\mathscr{S}$ = "x is always 0" is a (semi-informal) **software specification**.
  - It denotes **all imaginable softwares** with an x which is always 0:

$$[\![\mathscr{S}]\!] = \{S \mid \forall\, \sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots \in [\![S]\!]\ \forall\, i \in \mathbb{N}_0 \bullet \sigma_i \models x = 0\}$$

- The software specification "*true*" ("I don't care at all") denotes **chaos**.

A Bit More Abstract: Software vs. 'Real World' Requirements

In other words:

- **common sense understanding**: choose from "$\emptyset \cup requirements$"
- **software engineering understanding**:

  choose from "$chaos \cap requirements$"

- $\mathscr{S}$ = "x is always 0" is a (semi-informal) **software specification**.
  - It denotes **all imaginable softwares** with an x which is always 0:

$$[\![\mathscr{S}]\!] = \{S \mid \forall\, \sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots \in [\![S]\!]\ \forall\, i \in \mathbb{N}_0 \bullet \sigma_i \models x = 0\}$$

- The software specification "*true*" ("I don't care at all") denotes **chaos**.
- Writing a requirements specification means **constraining chaos**, or describing (a) subset(s) of chaos.
- **Design**/**Implementation** means: choosing from the obtained subset(s) of chaos.

And What's Hard About That?

Example: customer says "I need a computation of square numbers, i.e. $f : x \mapsto x^2$"

And What's Hard About That?

Example: customer says "I need a computation of square numbers, i.e. $f : x \mapsto x^2$"

- **We've got options** to choose from:

```
1  int sq( int x ) {
2    return x * x;
3  }
```

→ 1h, 100 €

```
1  unsigned int
2  sq( unsigned short x ) {
3    return x * x;
4  }
```

→ 2h, 200 €

```
1  #include <gmp.h>
2  void sq( mpz_t x ) {
3    mpz_mul( x, x, x );
4  }
```

→ 4h, 400 €

And What's Hard About That?

Example: customer says "I need a computation of square numbers, i.e. $f : x \mapsto x^2$"

- **We've got options** to choose from:

```
1  int sq( int x ) {
2    return x * x;
3  }
```

$\rightarrow$ 1h, 100 €

```
1  unsigned int
2  sq( unsigned short x ) {
3    return x * x;
4  }
```

$\rightarrow$ 2h, 200 €

```
1  #include <gmp.h>
2  void sq( mpz_t x ) {
3    mpz_mul( x, x, x );
4  }
```

$\rightarrow$ 4h, 400 €

with errors: $sq\left(2^{31}-1\right) = 1$

And What's Hard About That?

Example: customer says "I need a computation of square numbers, i.e. $f : x \mapsto x^2$"

- **We've got options** to choose from:

```
1  int sq( int x ) {
2    return x * x;
3  }
```

$\rightarrow$ 1h, 100 €

with errors: $sq\left(2^{31}-1\right) = 1$

```
1  unsigned int
2  sq( unsigned short x ) {
3    return x * x;
4  }
```

$\rightarrow$ 2h, 200 €

not defined for 2^{16}

```
1  #include <gmp.h>
2  void sq( mpz_t x ) {
3    mpz_mul( x, x, x );
4  }
```

$\rightarrow$ 4h, 400 €

And What's Hard About That?

Example: customer says "I need a computation of square numbers, i.e. $f : x \mapsto x^2$"

- **We've got options** to choose from:

```
1  int sq( int x ) {
2    return x * x;
3  }
```

→ 1h, 100 €

with errors: $sq\left(2^{31}-1\right) = 1$

```
1  unsigned int
2  sq( unsigned short x ) {
3    return x * x;
4  }
```

→ 2h, 200 €

not defined for 2^{16}

```
1  #include <gmp.h>
2  void sq( mpz_t x ) {
3    mpz_mul( x, x, x );
4  }
```

→ 4h, 400 €

usage non-trivial:

```
1  mpz_t x;
2  mpz_init(x);
3  mpz_set_si( x, 2147483647 );
4  sq(x);
5  fprintf( stdout,
6        "%i -> ", 2147483647 );
7  mpz_out_str( stdout, 10, x );
8  fprintf( stdout, "\n" );
```

And What's Hard About That?

Example: customer says "I need a computation of square numbers, i.e. $f : x \mapsto x^2$"

- **We've got options** to choose from:

```
1  int sq( int x ) {
2    return x * x;
3  }
```

→ 1h, 100 €

with errors: $sq\left(2^{31}-1\right) = 1$

```
1  unsigned int
2  sq( unsigned short x ) {
3    return x * x;
4  }
```

→ 2h, 200 €

not defined for 2^{16}

```
1  #include <gmp.h>
2  void sq( mpz_t x ) {
3    mpz_mul( x, x, x );
4  }
```

→ 4h, 400 €

usage non-trivial:

```
1  mpz_t x;
2  mpz_init(x);
3  mpz_set_si( x, 2147483647 );
4  sq(x);
5  fprintf( stdout,
6        "%i -> ", 2147483647 );
7  mpz_out_str( stdout, 10, x );
8  fprintf( stdout, "\n" );
```

- Okay, **customer said**: input values are from $\{0, \ldots, 27\}$.

And What's Hard About That?

Example: customer says "I need a computation of square numbers, i.e. $f : x \mapsto x^2$"

- **We've got options** to choose from:

```c
int sq( int x ) {
  return x * x;
}
```

→ 1h, 100 €

with errors: $sq\left(2^{31}-1\right) = 1$

```c
unsigned int
sq( unsigned short x ) {
  return x * x;
}
```

→ 2h, 200 €

not defined for 2^{16}

```c
#include <gmp.h>
void sq( mpz_t x ) {
  mpz_mul( x, x, x );
}
```

→ 4h, 400 €

usage non-trivial:

```c
mpz_t x;
mpz_init(x);
mpz_set_si( x, 2147483647 );
sq(x);
fprintf( stdout,
      "%i -> ", 2147483647 );
mpz_out_str( stdout, 10, x );
fprintf( stdout, "\n" );
```

- Okay, **customer said**: input values are from $\{0, \ldots, 27\}$.
- **Still**, we've got options. . . :

```c
unsigned int sq( unsigned short x ) {
  unsigned int r = 0, n;
  for (n = x; n; --n) {
    r += x; sleep(1);
  }
  return r;
}
```

A First Summary

A **good** requirements specification

(i) **avoids disputes** with the customer at (milestones or) delivery time,

(ii) prevents us from doing **too little/too much** work,

(iii) is economic (see below).

A First Summary

A **good** requirements specification

(i) **avoids disputes** with the customer at (milestones or) delivery time,

(ii) prevents us from doing **too little/too much** work,

(iii) is economic (see below).

There's a **tradeoff** between **narrowness** and **openness**:

A First Summary

A **good** requirements specification

(i) **avoids disputes** with the customer at (milestones or) delivery time,

(ii) prevents us from doing **too little/too much** work,

(iii) is economic (see below).

There's a **tradeoff** between **narrowness** and **openness**:

- the optimum wrt. (i) and (ii) is the complete, perfect software product as needed by the customer (most narrow):
  - no disputes; amount of work needed is clear.

  Drawback: hard/expensive to get, that's bad for (iii).

A First Summary

A **good** requirements specification

(i) **avoids disputes** with the customer at (milestones or) delivery time,

(ii) prevents us from doing **too little/too much** work,

(iii) is economic (see below).

There's a **tradeoff** between **narrowness** and **openness**:

- the optimum wrt. (i) and (ii) is the complete, perfect software product as needed by the customer (most narrow):
  - no disputes; amount of work needed is clear.

  Drawback: hard/expensive to get, that's bad for (iii).
- The cheapest (not: most economic) requirements specification is "do what you want" (most open).

  Drawback: high risk value for not doing (i) and (ii).

A First Summary

A **good** requirements specification

(i) **avoids disputes** with the customer at (milestones or) delivery time,

(ii) prevents us from doing **too little/too much** work,

(iii) is economic (see below).

There's a **tradeoff** between **narrowness** and **openness**:

- the optimum wrt. (i) and (ii) is the complete, perfect software product as needed by the customer (most narrow):
  - no disputes; amount of work needed is clear.

  Drawback: hard/expensive to get, that's bad for (iii).
- The cheapest (not: most economic) requirements specification is "do what you want" (most open).

  Drawback: high risk value for not doing (i) and (ii).
- Being too **narrow** has another severe **drawback**...

. . . Namely: Requirements Specification vs. Design

- Idealised views advocate a strict **separation** between **requirements** (“what is to be done?”) and **design** (“how are things done?”).
- **Rationale**: Fixing **too much** of the “how” **too early** may unnecessarily narrow down the design space and inhibit new ideas.

  There may be **better** (easier, cheaper, . . .) solutions than the one imagined first.

. . . Namely: Requirements Specification vs. Design

- Idealised views advocate a strict **separation** between **requirements** (“what is to be done?”) and **design** (“how are things done?”).
- **Rationale**: Fixing **too much** of the “how” **too early** may unnecessarily narrow down the design space and inhibit new ideas.

  There may be **better** (easier, cheaper, . . .) solutions than the one imagined first.

- In practice, this separation is often neither possible nor advisable:
  - Existing components should be considered. ($\rightarrow$ **re-use**)
  - Customer, (safety) regulations, norms, etc. may **require a particular solution** anyway.
  - It is often useful to **reflect real-world structures** in the software.
  - In some (low risk) cases it may be **more economical** to skip requirements analysis and directly code (and document!) a proposal.
  - Complex systems may need a **preliminary system design** before being able to understand and describe requirements.
  - One way to **check** a requirements specification for consistency (realisability) is to think through at least one possible solution.
  - The requirements specification should answer **questions** arising during development.

Requirements Engineering: Basics

Requirements Analysis Basics

- **Note**: analysis in the sense of **"finding out what the exact requirements are"**.
  "Analysing an existing requirements/feature specification" → later.

In the following we shall discuss:

(i) **documents** of the requirements analysis:
- dictionary,
- requirements/feature specification.

(ii) desired **properties** of
- requirements specifications,
- requirements specification documents,

(iii) **kinds** of requirements
- hard and soft,
- open and tacit,
- functional and non-functional, `*sigh*`

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

Documents of the Requirements Analysis: Dictionary

- Requirements analysis should be based on a **dictionary**.

Documents of the Requirements Analysis: 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.

Documents of the Requirements Analysis: 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 **seem** "crystal clear" at first sight are **not uncommon**.

Documents of the Requirements Analysis: 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 **seem** "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).

Documents of the Requirements Analysis: 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.

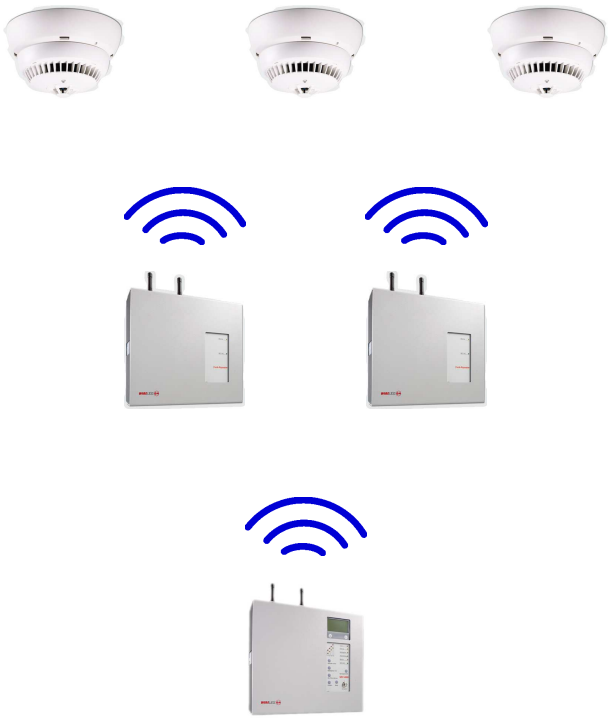
  Note: entries for terms that **seem** "crystal clear" at first sight are **not uncommon**.
- All work on requirements should, as far as possible, be done **using terms from the dictionary** consistently and consequently.

  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 (With Room for Improvement)

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** (wireless sensors and manual indicators),
  - **a central unit** (not a participant),
  - and **repeaters** (a non-terminal 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)

Dictionary Example (With Room for Improvement)

Example: Wireless Fire Alarm System

- During a
  EN-54-2
  protocol
  compone
  - termin
    (wirele
  - a cent
  - and re
- repeaters
  similar, l
  requirem

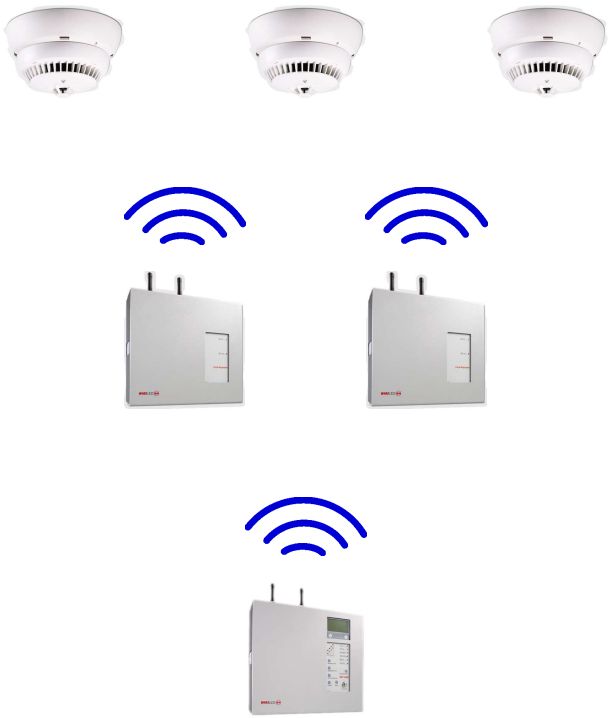
Excerpt from the dictionary (ca. 50 entries):

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**. A repeater consists of the following **devices**: wireless communication module (1 to 4, here fixed to 2), [...]

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. A central unit consist of the following **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.



(Arenis et al., 2014)

Documents of the Requirements Analysis: Specifications

- **Recall**:

  Lastenheft (Requirements Specification) Entire demands on deliverables and services of a developer within a contracted development, **created by the customer**.

  Pflichtenheft (Feature Specification) Specification of how to realise a given requirements specification, **created by the developer**.

  DIN 69901-5 (2009)

Documents of the Requirements Analysis: Specifications

- **Recall**:

  Lastenheft (Requirements Specification) Entire demands on deliverables and services of a developer within a contracted development, **created by the customer**.

  Pflichtenheft (Feature Specification) Specification of how to realise a given requirements specification, **created by the developer**.

  DIN 69901-5 (2009)

- **Recommendation**:

  If the **requirements specification** is **not given** by customer (but needs to be developed), focus on and maintain only the **collection of requirements** (possibly sketchy and unsorted) and maintain the **feature specification**.

  (Ludewig and Lichter, 2013)

Documents of the Requirements Analysis: Specifications

- **Recall**:

  Lastenheft (Requirements Specification) Entire demands on deliverables and services of a developer within a contracted development, **created by the customer**.

  Pflichtenheft (Feature Specification) Specification of how to realise a given requirements specification, **created by the developer**.

  DIN 69901-5 (2009)

- **Recommendation**:

  > If the **requirements specification** is **not given** by customer (but needs to be developed), focus on and maintain only the **collection of requirements** (possibly sketchy and unsorted) and maintain the **feature specification**.
  >
  > (Ludewig and Lichter, 2013)

- **Note**: In the following (unless otherwise noted), we discuss the **feature specification**, i.e. the basis of the software development.

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

Requirements on Requirements Specifications

A **requirements specification** should be

Requirements on Requirements Specifications

A **requirements specification** should be

- **correct**
  — it correctly represents the wishes/needs of the customer,

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,

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,

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**,

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,

Requirements on Requirements Specifications

A **requirements specification** should be

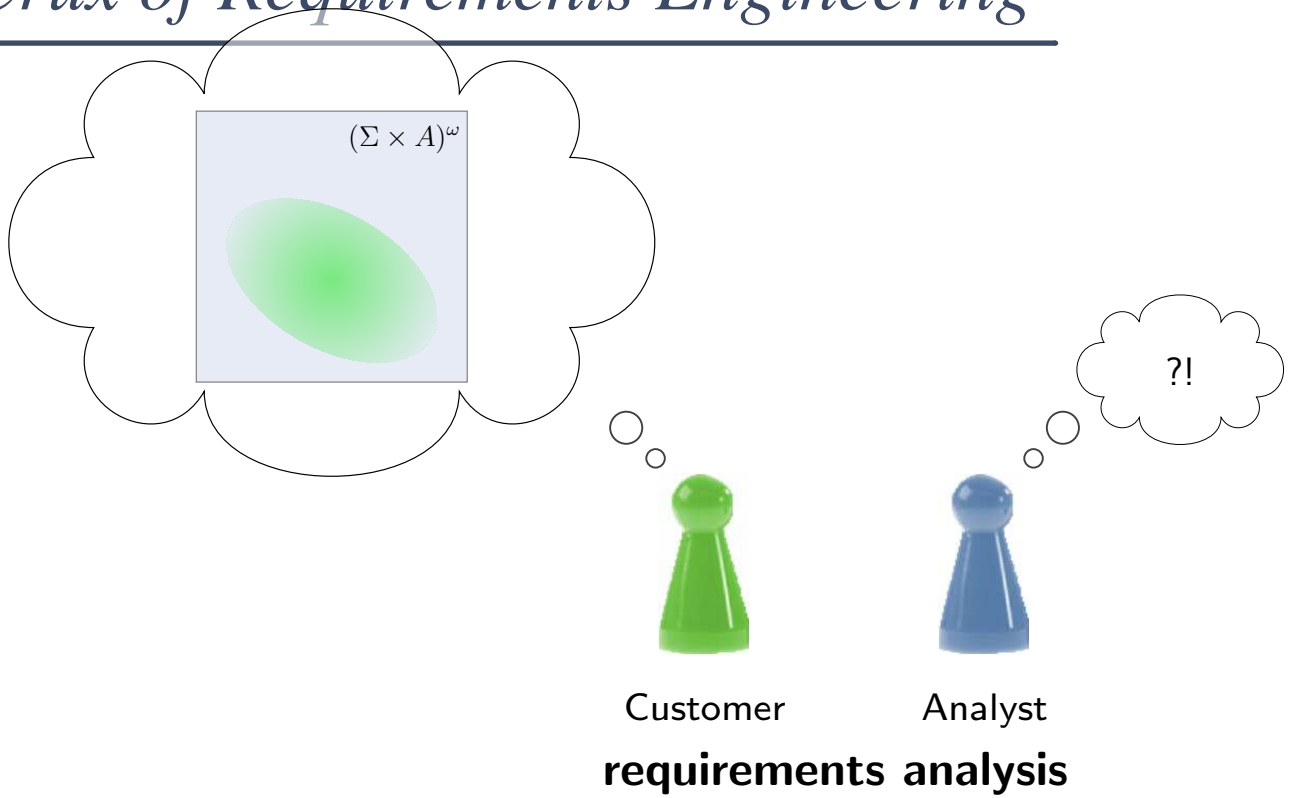
- **correct**
  — it correctly represents the wishes/needs of the customer,
- **complete**
  — all requirements (existing in somebody's head, or a document, or . . .) should be present,
- **relevant**
  — things which are not relevant to the project should not be constrained,
- **consistent**, **free 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,

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.

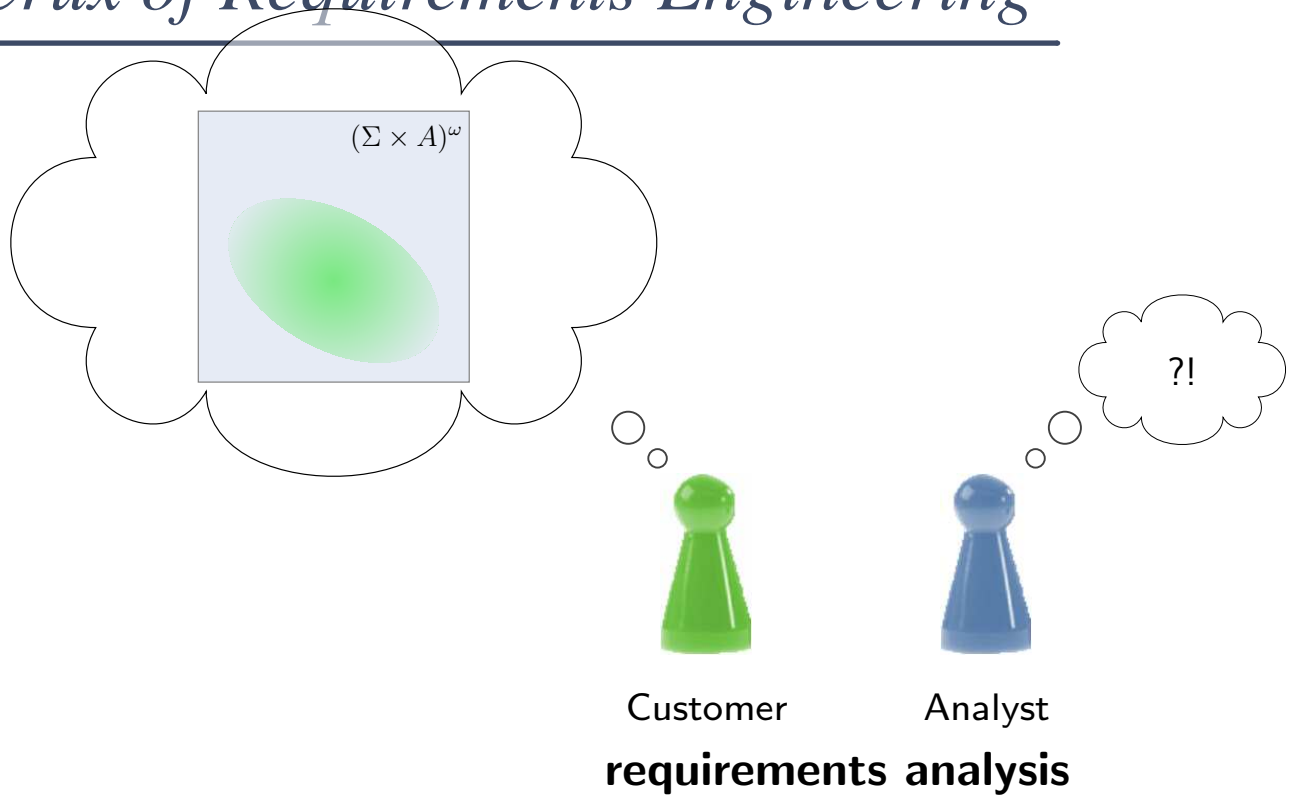
The Crux of Requirements Engineering



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

  $\rightarrow$ in that case, there is **hardly a chance** to be sure of **correctness** and **completeness**.

The Crux of Requirements Engineering



requirements analysis

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

  $\rightarrow$ in that case, there is **hardly a chance** to be sure of **correctness** and **completeness**.
- **"Dear customer, please tell me/write down 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:

Requirements on Requirements Specification Documents

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

- **easily understandable**, **not unnecessarily complicated** —
  all affected people are able to understand the requirements specification,

Requirements on Requirements Specification Documents

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

- **easily understandable**, **not unnecessarily complicated** —
  all affected people are able to understand the requirements specification,
- **precise** —
  the requirements specification does not introduce new unclarities or rooms for interpretation ($\rightarrow$ testable, objective),

Requirements on Requirements Specification Documents

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

- **easily understandable**, **not unnecessarily complicated** —
  all affected people are able to understand the requirements specification,
- **precise** —
  the requirements specification does 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,

Requirements on Requirements Specification Documents

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

- **easily understandable**, **not unnecessarily complicated** —
  all affected people are able to understand the requirements specification,
- **precise** —
  the requirements specification does 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.

Requirements on Requirements Specification Documents

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

- **easily understandable**, **not unnecessarily complicated** —
  all affected people are able to understand the requirements specification,
- **precise** —
  the requirements specification does 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 easy 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** (most changes require reading beforehand).

Kinds of Requirements

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.

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 the 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.

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 the 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:**

- 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.

→ 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, not considered relevant by the customer, not known to be relevant).

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, 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 side because our main users are socialised with right-to-left reading direction,
  - the ECU (embedded control unit) may only use a certain amount of bus capacity.

- distinguish **don't care**:

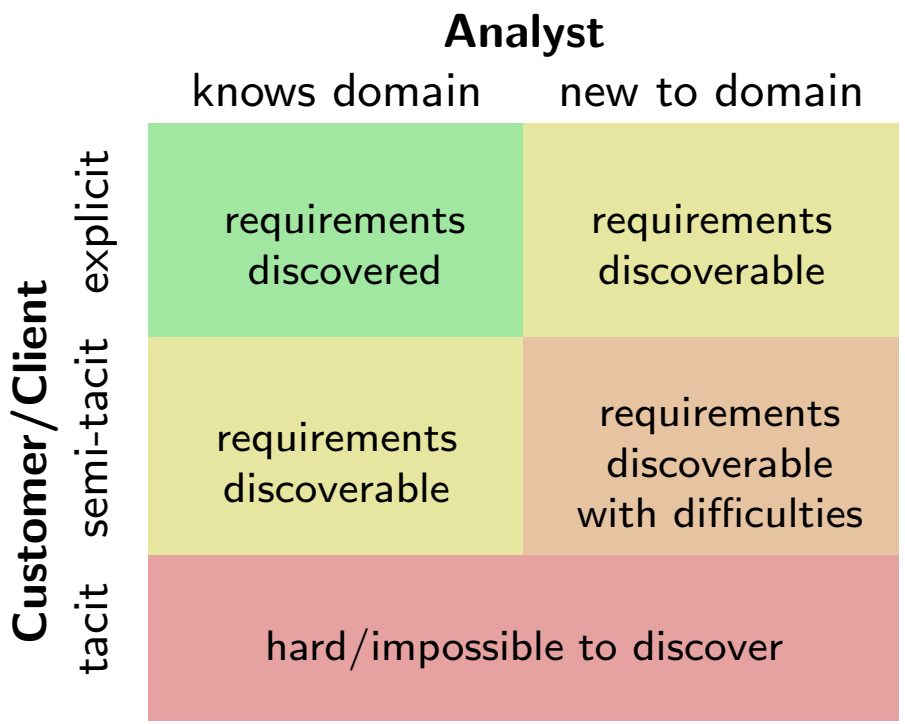
  intentionally left to be decided by developer.

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, 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 side because our main users are socialised with right-to-left reading direction,
  - the ECU (embedded control unit) may only use a certain amount of bus capacity.
- distinguish **don't care**:

  intentionally left to be decided by developer.



(Gacitua et al., 2009)

Kinds of Requirements: Functional and Non-Functional

- `*sigh*`

Kinds of Requirements: Functional and Non-Functional

- `*sigh*`
- **Recall definition of software**:

  A finite description of a set of computation paths of the form

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

  Note: states σ may be labelled with timestamps, or energy consumption so far, ...
- **Another view**: software is a function which maps input to output sequences:

  $$S : \sigma_0^i \xrightarrow{\alpha_1^i} \sigma_1^i \xrightarrow{\alpha_2^i} \sigma_2^i \cdots \mapsto \begin{pmatrix} \sigma_0^i \\ \sigma_0^o \end{pmatrix} \xrightarrow[\alpha_1^o]{\alpha_1^i} \begin{pmatrix} \sigma_1^i \\ \sigma_1^o \end{pmatrix} \xrightarrow[\alpha_2^o]{\alpha_2^i} \cdots$$

Kinds of Requirements: Functional and Non-Functional

- `*sigh*`
- **Recall definition of software**:

  A finite description of a set of computation paths of the form

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

  Note: states σ may be labelled with timestamps, or energy consumption so far, ...
- **Another view**: software is a function which maps input to output sequences:

  $$S : \sigma_0^i \xrightarrow{\alpha_1^i} \sigma_1^i \xrightarrow{\alpha_2^i} \sigma_2^i \cdots \mapsto \begin{pmatrix} \sigma_0^i \\ \sigma_0^o \end{pmatrix} \xrightarrow[\alpha_1^o]{\alpha_1^i} \begin{pmatrix} \sigma_1^i \\ \sigma_1^o \end{pmatrix} \xrightarrow[\alpha_2^o]{\alpha_2^i} \cdots$$

- **Every constraint** on things **observable** in the computation paths 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...

Requirements Analysis Techniques

(A Selection of) Analysis Techniques

Analysis Technique	**Focus** current situation	desired situation	innovation consequences
Analysis of existing data and documents	■		
Observation	■		
Questionning with $\left(\begin{array}{c}\text{closed}\\ \text{structured}\\ \text{open}\end{array}\right)$ questions	■	■	
Interview		■	
Modelling		■	
Experiments		■	
Prototyping		■	
Participative development		■	■

(Ludewig and Lichter, 2013)

Requirements Elicitation

- **Observation**:

  Customers are typically not trained in stating/communicating requirements.
  They live in the **"I want a pony"**-world — in multiple senses. . . ;-)

Requirements Elicitation

- **Observation**:

  Customers are typically not trained in stating/communicating requirements.
  They live in the **"I want a pony"**-world — in multiple senses. . . ;-)

- 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.

Requirements Elicitation

- **Observation**:

  Customers are typically not trained in stating/communicating requirements.
  They live in the **"I want a pony"**-world — in multiple senses. . . ;-)

- 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.

A **made up** dialogue:

Analyst: *So in the morning, you open the door at the main entrance?*

Requirements Elicitation

- **Observation**:

  Customers are typically not trained in stating/communicating requirements.
  They live in the **"I want a pony"**-world — in multiple senses. . . ;-)

- 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.

A **made up** dialogue:

Analyst: *So in the morning, you open the door at the main entrance?*

Customer: *Yes, as I told you.*

Requirements Elicitation

- **Observation**:

  Customers are typically not trained in stating/communicating requirements.
  They live in the **"I want a pony"**-world — in multiple senses. . . ;-)

- 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.

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?*

Requirements Elicitation

- **Observation**:

  Customers are typically not trained in stating/communicating requirements.
  They live in the **"I want a pony"**-world — in multiple senses. . . ;-)

- 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.

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.*

Requirements Elicitation

- **Observation**:

  Customers are typically not trained in stating/communicating requirements.
  They live in the **"I want a pony"**-world — in multiple senses. . . ;-)

- 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.

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?*

Requirements Elicitation

- **Observation**:

  Customers are typically not trained in stating/communicating requirements.
  They live in the **"I want a pony"**-world — in multiple senses. . . ;-)

- 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.

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.*

Requirements Elicitation

- **Observation**:

  Customers are typically not trained in stating/communicating requirements.
  They live in the **"I want a pony"**-world — in multiple senses. . . ;-)

- 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.

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?*

Requirements Elicitation

- **Observation**:

  Customers are typically not trained in stating/communicating requirements.
  They live in the **"I want a pony"**-world — in multiple senses. . . ;-)

- 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.

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.*

Requirements Elicitation

- **Observation**:

  Customers are typically not trained in stating/communicating requirements.
  They live in the **"I want a pony"**-world — in multiple senses. . . ;-)

- 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.

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?*

Requirements Elicitation

- **Observation**:

  Customers are typically not trained in stating/communicating requirements.
  They live in the **"I want a pony"**-world — in multiple senses. . . ;-)

- 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.

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.*

Requirements Elicitation

- **Observation**:

  Customers are typically not trained in stating/communicating requirements.
  They live in the **"I want a pony"**-world — in multiple senses. . . ;-)

- 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.

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?*

Requirements Elicitation

- **Observation**:

  Customers are typically not trained in stating/communicating requirements.
  They live in the **"I want a pony"**-world — in multiple senses. . . ;-)

- 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.

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.*

Requirements Elicitation

- **Observation**:

  Customers are typically not trained in stating/communicating requirements.
  They live in the **"I want a pony"**-world — in multiple senses. . . ;-)

- 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.

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. One searches 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**.

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. One searches 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're things done today? What should be improved?

Specification Languages

Requirements Specification Language

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)

Requirements Specification Language

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)

specification language — A language, often a machine-processable 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

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)**

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)**

Requirements Specification Language

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)**

specification language — A language, often a machine-processable 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)**

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)**

Requirements Specification Language

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)**

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)**

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)**

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	Nouns like "registration" often hide complex processes

	quantifiers.	"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 written using A, B, C, D, E, F where

A	clarifies when and under what conditions the activity takes place

Natural Language Patterns

Natural language requirements can be written using A, B, C, D, E, F 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)

Natural Language Patterns

Natural language requirements can be written using A, B, C, D, E, F 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

Natural Language Patterns

Natural language requirements can be written using A, B, C, D, E, F 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

Natural Language Patterns

Natural language requirements can be written using A, B, C, D, E, F 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

Natural Language Patterns

Natural language requirements can be written using A, B, C, D, E, F 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)

Natural Language Patterns

Natural language requirements can be written using A, B, C, D, E, F 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.

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

6. Guidance in the use of these Imperatives

Imperatives of the type defined in this memo must be use and sparingly. In particular, they MUST only be used wh actually required for interoperation or to limit behavio potential for causing harm (e.g., limiting retransmisssi example, they must not be used to try to impose a partic 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 implementi SHOULD, or doing something the specification says MUST N NOT be done may be very subtle. Document authors should to elaborate the security implications of not following recommendations or requirements as most implementors wil had the benefit of the experience and discussion that pr specification.

8. Acknowledgments

The definitions of these terms are an amalgam of definit from a number of RFCs. In addition, suggestions have be incorporated from a number of people including Robert Ul Narten, Neal McBurnett, and Robert Elz.

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
- **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)

You Are Here

Introduction	L 1:	20.4., Mo
	T 1:	23.4., Do
Development Process, Metrics	L 2:	27.4., Mo
	L 3:	30.4., Do
	L 4:	4.5., Mo
	T 2:	7.5., Do
Requirements Engineering	L 5:	11.5., Mo
	-	14.5., Do
	L 6:	18.5., Mo
	L 7:	21.5., Do
	-	25.5., Mo
	-	28.5., Do
	T 3:	1.6., Mo
Design Modelling & Analysis	-	4.6., Do
	L 8:	8.6., Mo
	L 9:	11.6., Do
	L 10:	15.6., Mo
	T 4:	18.6., Do
Implementation, Testing	L 11:	22.6., Mo
	L 12:	25.6., Do
	L 13:	29.6., Mo
	T 5:	2.7., Do
Formal Verification	L 14:	6.7., Mo
	L 15:	9.7., Do
	L 16:	13.7., Mo
	T 6:	16.7., Do
The Rest	L 17:	20.7., Mo
	L 18:	23.7., Do

Recall: Formal Methods

Formal Methods (in the Software Development Domain)

... back to "'technological paradise' where 'no acts of God can be permitted' and everything happens according to the blueprints".

(Kopetz, 2011; Lovins and Lovins, 2001)

Definition. [Bjørner and Havelund (2014)]
A method is called **formal method** if and only if its techniques and tools can be explained in mathematics.

Example: If a method includes, as a tool, a specification language, then that language has

- a **formal syntax**,
- a **formal semantics**, and
- a **formal proof system**.

– 1 – 2015-04-20 – main –

Formal, Rigorous, or Systematic Development

"The **techniques** of a formal method help

- **construct** a specification, and/or
- **analyse** a specification, and/or
- **transform** (**refine**) one (or more) specification(s) into a **program**.

The **techniques** of a formal method, (besides the specification languages) are typically software packages that help developers use the techniques and other tools.

The aim of developing software, either

- **formally** (all arguments are formal) or
- **rigorously** (some arguments are made and they are formal) or
- **systematically** (some arguments are made on a form that can be made formal)

is to (be able to) **reason in a precise manner about properties** of what is being developed." (Bjørner and Havelund, 2014)

21/37

– 1 – 2015-04-20 – main –

Recall: Formal Methods

Formal Methods (in the Software Development Domain)

... back to "'technological paradise' where 'no acts of God can be permitted' and everything happens according to the blueprints".

(Kopetz, 2011; Lovins and Lovins, 2001)

Definition. [Bjørner and Havelund (2014)]
A method is called **formal method** if and only if its techniques and tools can be explained in mathematics.

Example: If a method includes, as a tool, a specification language, then that language has

- a **formal syntax**,
- a **formal semantics**, and
- a **formal proof system**.

– 1 – 2015-04-20 – main –

Formal, Rigorous, or Systematic Development

"The **techniques** of a formal method help

- **construct** a specification, and/or
- **analyse** a specification, and/or
- **transform** (**refine**) one (or more) specification(s) into a **program**.

The **techniques** of a formal method, (besides the specification languages) are typically software packages that help developers use the techniques and other tools.

The aim of developing software, either

- **formally** (all arguments are formal) or
- **rigorously** (some arguments are made and they are formal) or
- **systematically** (some arguments are made on a form that can be made formal)

is to (be able to) **reason in a precise manner about properties** of what is being developed." (Bjørner and Havelund, 2014)

21/37

– 1 – 2015-04-20 – main –

Decision tables (DT) are **one example** for a formal requirements specification language:

- we give a formal syntax and semantics,
- **requirements quality criteria**, e.g. completeness, can be **formally defined**,
- **thus** for a DT we can **formally argue** whether it is complete or not,
- (some) formal arguments can be done **automatically** (→ tool support).

Formal Specification and Analysis of Requirements: Decision Tables for Example

Decision Tables

Definition. [*Decision Table*] Let C be a set of (atomic) **conditions** and A a set of **actions**.

(i) The set $\Phi(C)$ of **premises** over C consists of the terms defined by the following grammar: $\varphi ::= \mathit{true} \mid c \mid \neg\varphi_1 \mid \varphi_1 \vee \varphi_2$, $c \in C$.

(ii) A **rule** r over C and A is a pair (φ, α), denoted by $\varphi \to \alpha$, which comprises a premise $\varphi \in \Phi(C)$ and a finite set $\alpha \subseteq A$ of actions (the **effect**).

(iii) Any finite set T of rules over C and A is called **decision table** (over C and A).

Decision Tables: Example

This might've been the specification of lecture hall 101-0-026's **ventilation system**:

- Conditions:

  $C = \{button\_pressed, ventilation\_on, ventilation\_off\}$ **shorthands**: $\{b, on, off\}$.
- Actions:

  $A = \{do\_ventilate, stop\_ventilate\}$ **shorthands**: $\{go, stop\}$.
- Rules:
  - $r_1 = b \wedge ventilation\_off \rightarrow \{go\}$
  - $r_2 = b \wedge ventilation\_on \rightarrow \{stop\}$
- Decision table:

  $T = \{r_1, r_2\}$.

Satisfaction of Conditions and Premises

- Let Σ be a set of **states** with a **satisfaction relation** $\models_0 \subseteq \Sigma \times C$.

  We say σ **satisfies** condition c, and write $\sigma \models_0 c$, if and only if $(\sigma, c) \in \models_0$, otherwise we write $\sigma \not\models_0 \varphi$.

Satisfaction of Conditions and Premises

- Let Σ be a set of **states** with a **satisfaction relation** $\models_0 \subseteq \Sigma \times C$.

  We say σ **satisfies** condition c, and write $\sigma \models_0 c$, if and only if $(\sigma, c) \in \models_0$, otherwise we write $\sigma \not\models_0 \varphi$.

Example: (useful sets of states for room ventilation model)

- $\Sigma_1 = C \to \{0,1\} (= \mathbb{B})$ — a state $\sigma \in \Sigma_1$ is a (boolean) valuation of conditions

  $\models_0$ defined by: $\sigma \models_0 c$ if and only if $\sigma(c) = 1$.

Satisfaction of Conditions and Premises

- Let Σ be a set of **states** with a **satisfaction relation** $\models_0 \subseteq \Sigma \times C$.

  We say σ **satisfies** condition c, and write $\sigma \models_0 c$, if and only if $(\sigma, c) \in \models_0$, otherwise we write $\sigma \not\models_0 \varphi$.

Example: (useful sets of states for room ventilation model)

- $\Sigma_1 = C \to \{0, 1\} (= \mathbb{B})$ — a state $\sigma \in \Sigma_1$ is a (boolean) valuation of conditions

  $\models_0$ defined by: $\sigma \models_0 c$ if and only if $\sigma(c) = 1$.

- $\Sigma_2 = \{b, V\} \to \mathbb{B} \cup \mathbb{R}_0^+$; $\sigma(b) \in \mathbb{B}$ (**button state**), $\sigma(V) \in \mathbb{R}_0^+$ (**voltage at ventilator**)

  $\models_0$ defined by:

  - $\sigma \models_0 b$ if and only if $\sigma(b) = 1$,
  - $\sigma \models_0 on$ if and only if $\sigma(V) \geq 0.7$, (ventilator rotates),
  - $\sigma \models_0 off$ if and only if $\sigma(V) < 0.7$. (voltage too low for rotation),

Satisfaction of Conditions and Premises

- Let Σ be a set of **states** with a **satisfaction relation** $\models_0 \subseteq \Sigma \times C$.

  We say σ **satisfies** condition c, and write $\sigma \models_0 c$, if and only if $(\sigma, c) \in \models_0$, otherwise we write $\sigma \not\models_0 \varphi$.

Example: (useful sets of states for room ventilation model)

- $\Sigma_1 = C \to \{0,1\} (= \mathbb{B})$ — a state $\sigma \in \Sigma_1$ is a (boolean) valuation of conditions

  $\models_0$ defined by: $\sigma \models_0 c$ if and only if $\sigma(c) = 1$.

- $\Sigma_2 = \{b, V\} \to \mathbb{B} \cup \mathbb{R}_0^+$; $\sigma(b) \in \mathbb{B}$ (**button state**), $\sigma(V) \in \mathbb{R}_0^+$ (**voltage at ventilator**)

  $\models_0$ defined by:

  - $\sigma \models_0 b$ if and only if $\sigma(b) = 1$,
  - $\sigma \models_0 on$ if and only if $\sigma(V) \geq 0.7$, (ventilator rotates),
  - $\sigma \models_0 off$ if and only if $\sigma(V) < 0.7$. (voltage too low for rotation),

- **In other words**: Σ can be **whatever you want**, as long as you explain/define how to read out from $\sigma \in \Sigma$ whether conditions b, on, off should be considered satisfied in σ.

Satisfaction of Premises

- **Given** $(\Sigma, \models_0)$, the usual interpretation of the logical connectives

$$\mathit{true}, \neg, \vee$$

induces a satisfaction relation $\models \subseteq \Sigma \times \Phi(C)$ for **premises** as follows:

- $\sigma \models \mathit{true}$, for all $\sigma \in \Sigma$,
- $\sigma \models \neg c$, if and only if $\sigma \not\models_0 c$,
- $\sigma \models c_1 \vee c_2$, if and only if $\sigma \models_0 c_1$ or $\sigma \models_0 c_2$.

Satisfaction of Premises

- **Given** $(\Sigma, \models_0)$, the usual interpretation of the logical connectives

$$\mathit{true}, \neg, \vee$$

induces a satisfaction relation $\models \subseteq \Sigma \times \Phi(C)$ for **premises** as follows:

  - $\sigma \models \mathit{true}$, for all $\sigma \in \Sigma$,
  - $\sigma \models \neg c$, if and only if $\sigma \not\models_0 c$,
  - $\sigma \models c_1 \vee c_2$, if and only if $\sigma \models_0 c_1$ or $\sigma \models_0 c_2$.

- We call $r = \varphi \to \alpha$ over C and A **enabled** σ if and only if $\sigma \models \varphi$.

Note: In the following, we may use $\wedge$, $\Longrightarrow$, $\Longleftrightarrow$ as abbreviations as usual.

Satisfaction of Premises

- **Given** $(\Sigma, \models_0)$, the usual interpretation of the logical connectives

$$\mathit{true}, \neg, \vee$$

induces a satisfaction relation $\models \subseteq \Sigma \times \Phi(C)$ for **premises** as follows:

  - $\sigma \models \mathit{true}$, for all $\sigma \in \Sigma$,
  - $\sigma \models \neg c$, if and only if $\sigma \not\models_0 c$,
  - $\sigma \models c_1 \vee c_2$, if and only if $\sigma \models_0 c_1$ or $\sigma \models_0 c_2$.

- We call $r = \varphi \to \alpha$ over C and A **enabled** σ if and only if $\sigma \models \varphi$.

Note: In the following, we may use $\wedge$, $\Longrightarrow$, $\Longleftrightarrow$ as abbreviations as usual.

Example:

- Consider $\sigma \in \Sigma_1$ with $\sigma = \{b \mapsto 1, on \mapsto 1, \mathit{off} \mapsto 0\}$ and $\varphi = b \wedge on \in \Phi(C)$.

  Then $\sigma \models \varphi$, thus r is enabled in σ.

Specifying Behaviour with Decision Tables

A decision table T **induces computation paths** as follows:

- Let C be a set of conditions, A a set of actions.
- Let $(\Sigma, \models_0)$ be a set of **states** with a **satisfaction relation** for conditions.
- Set $\mathcal{A} := 2^A$ as **event** set.

Specifying Behaviour with Decision Tables

A decision table T **induces computation paths** as follows:

- Let C be a set of conditions, A a set of actions.
- Let $(\Sigma, \models_0)$ be a set of **states** with a **satisfaction relation** for conditions.
- Set $\mathcal{A} := 2^A$ as **event** set.

A (finite or infinite) state/event sequence

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

is called **computation path** of T if and only if

- for all $i \in \mathbb{N}_0$, exists $r = \varphi \to \alpha \in T$, such that $\sigma_i \models \varphi$ and $\alpha_{i+1} = \alpha$.

Specifying Behaviour with Decision Tables

A decision table T **induces computation paths** as follows:

- Let C be a set of conditions, A a set of actions.
- Let $(\Sigma, \models_0)$ be a set of **states** with a **satisfaction relation** for conditions.
- Set $\mathcal{A} := 2^A$ as **event** set.

A (finite or infinite) state/event sequence

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

is called **computation path** of T if and only if

- for all $i \in \mathbb{N}_0$, exists $r = \varphi \to \alpha \in T$, such that $\sigma_i \models \varphi$ and $\alpha_{i+1} = \alpha$.

  In other "words": if and only if $\forall\, i \in \mathbb{N}_0\ \exists\, r = \varphi \to \alpha \in T \bullet \sigma_i \models \varphi \wedge \alpha_{i+1} = \alpha$.

Specifying Behaviour with Decision Tables

A decision table T **induces computation paths** as follows:

- Let C be a set of conditions, A a set of actions.
- Let $(\Sigma, \models_0)$ be a set of **states** with a **satisfaction relation** for conditions.
- Set $\mathcal{A} := 2^A$ as **event** set.

A (finite or infinite) state/event sequence

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

is called **computation path** of T if and only if

- for all $i \in \mathbb{N}_0$, exists $r = \varphi \to \alpha \in T$, such that $\sigma_i \models \varphi$ and $\alpha_{i+1} = \alpha$.

  In other "words": if and only if $\forall\, i \in \mathbb{N}_0\ \exists\, r = \varphi \to \alpha \in T \bullet \sigma_i \models \varphi \wedge \alpha_{i+1} = \alpha$.

- Set $[\![T]\!]_{interleave} := \{\pi \mid \pi \text{ is a computation path of } T\}$.
  We call $[\![\cdot]\!]_{interleave}$ the **interleaving** interpretation/semantics of decision tables.

- **Note**: We will also define the **collecting** semantics $[\![\cdot]\!]_{collect}$ later.

Decision Tables: Example

Back to lecture hall 101-0-026's **ventilation system**:

- $C = \{button\_pressed, ventilation\_on, ventilation\_off\}$ **shorthands**: $\{b, on, off\}$.
- $A = \{do\_ventilate, stop\_ventilate\}$ **shorthands**: $\{go, stop\}$.
- $r_1 = b \wedge ventilation\_off \rightarrow \{go\}$
- $r_2 = b \wedge ventilation\_on \rightarrow \{stop\}$
- $T = \{r_1, r_2\}$.

What's in $[\![T]\!]_{interleaving}$?

Decision Tables: Example

Back to lecture hall 101-0-026's **ventilation system**:

- $C = \{button\_pressed, ventilation\_on, ventilation\_off\}$ **shorthands**: $\{b, on, off\}$.
- $A = \{do\_ventilate, stop\_ventilate\}$ **shorthands**: $\{go, stop\}$.
- $r_1 = b \wedge ventilation\_off \rightarrow \{go\}$
- $r_2 = b \wedge ventilation\_on \rightarrow \{stop\}$
- $T = \{r_1, r_2\}$.

What's in $[\![T]\!]_{interleaving}$? Naja, for **example**

- $\pi_1 = \sigma_0 \xrightarrow{\{go\}} \sigma_1 \xrightarrow{\{stop\}} \sigma_2$
  $\sigma_0 \models b \wedge off$, $\sigma_1 \models b \wedge on$

Decision Tables: Example

Back to lecture hall 101-0-026's **ventilation system**:

- $C = \{button\_pressed, ventilation\_on, ventilation\_off\}$ **shorthands**: $\{b, on, off\}$.
- $A = \{do\_ventilate, stop\_ventilate\}$ **shorthands**: $\{go, stop\}$.
- $r_1 = b \wedge ventilation\_off \rightarrow \{go\}$
- $r_2 = b \wedge ventilation\_on \rightarrow \{stop\}$
- $T = \{r_1, r_2\}$.

What's in $[\![T]\!]_{interleaving}$? Naja, for **example**

- $\pi_1 = \sigma_0 \xrightarrow{\{go\}} \sigma_1 \xrightarrow{\{stop\}} \sigma_2$
  $\sigma_0 \models b \wedge off$, $\sigma_1 \models b \wedge on$
- $\pi_2 = \sigma_0$
  $\sigma_0 \not\models b \wedge on$ and $\sigma_0 \not\models b \wedge off$

Decision Tables: Example

Back to lecture hall 101-0-026's **ventilation system**:

- $C = \{button\_pressed, ventilation\_on, ventilation\_off\}$ **shorthands**: $\{b, on, off\}$.
- $A = \{do\_ventilate, stop\_ventilate\}$ **shorthands**: $\{go, stop\}$.
- $r_1 = b \wedge ventilation\_off \rightarrow \{go\}$
- $r_2 = b \wedge ventilation\_on \rightarrow \{stop\}$
- $T = \{r_1, r_2\}$.

What's in $[\![T]\!]_{interleaving}$? Naja, for **example**

- $\pi_1 = \sigma_0 \xrightarrow{\{go\}} \sigma_1 \xrightarrow{\{stop\}} \sigma_2$
  $\sigma_0 \models b \wedge off$, $\sigma_1 \models b \wedge on$
- $\pi_2 = \sigma_0$
  $\sigma_0 \not\models b \wedge on$ and $\sigma_0 \not\models b \wedge off$
- and also $\pi_3 = \sigma_0 \xrightarrow{\{go\}} \sigma_1 \xrightarrow{\{go\}} \sigma_2$
  $\sigma_0 \models b \wedge off$, $\sigma_1 \models b \wedge off$

Decision Tables: Example

Back to lecture hall 101-0-026's **ventilation system**:

- $C = \{button\_pressed, ventilation\_on, ventilation\_off\}$ **shorthands**: $\{b, on, off\}$.
- $A = \{do\_ventilate, stop\_ventilate\}$ **shorthands**: $\{go, stop\}$.
- $r_1 = b \wedge ventilation\_off \rightarrow \{go\}$
- $r_2 = b \wedge ventilation\_on \rightarrow \{stop\}$
- $T = \{r_1, r_2\}$.

What's in $[\![T]\!]_{interleaving}$? Naja, for **example**

- $\pi_1 = \sigma_0 \xrightarrow{\{go\}} \sigma_1 \xrightarrow{\{stop\}} \sigma_2$
  $\sigma_0 \models b \wedge off$, $\sigma_1 \models b \wedge on$
- $\pi_2 = \sigma_0$
  $\sigma_0 \not\models b \wedge on$ and $\sigma_0 \not\models b \wedge off$
- and also $\pi_3 = \sigma_0 \xrightarrow{\{go\}} \sigma_1 \xrightarrow{\{go\}} \sigma_2$
  $\sigma_0 \models b \wedge off$, $\sigma_1 \models b \wedge off$
- also $\pi_4 = \sigma_0 \xrightarrow{\{go\}} \sigma_1 \xrightarrow{\{go\}} \sigma_2 \ldots$
  $\sigma_i \models b \wedge off$, $i \in \mathbb{N}_0$

Decision Tables: Example

Back to lecture hall 101-0-026's **ventilation system**:

- $C = \{button\_pressed, ventilation\_on, ventilation\_off\}$ **shorthands**: $\{b, on, off\}$.
- $A = \{do\_ventilate, stop\_ventilate\}$ **shorthands**: $\{go, stop\}$.
- $r_1 = b \wedge ventilation\_off \rightarrow \{go\}$
- $r_2 = b \wedge ventilation\_on \rightarrow \{stop\}$
- $T = \{r_1, r_2\}$.

What's in $[\![T]\!]_{interleaving}$? Naja, for **example**

- $\pi_1 = \sigma_0 \xrightarrow{\{go\}} \sigma_1 \xrightarrow{\{stop\}} \sigma_2$
  $\sigma_0 \models b \wedge off$, $\sigma_1 \models b \wedge on$
- $\pi_2 = \sigma_0$
  $\sigma_0 \not\models b \wedge on$ and $\sigma_0 \not\models b \wedge off$
- and also $\pi_3 = \sigma_0 \xrightarrow{\{go\}} \sigma_1 \xrightarrow{\{go\}} \sigma_2$
  $\sigma_0 \models b \wedge off$, $\sigma_1 \models b \wedge off$
- also $\pi_4 = \sigma_0 \xrightarrow{\{go\}} \sigma_1 \xrightarrow{\{go\}} \sigma_2 \ldots$
  $\sigma_i \models b \wedge off$, $i \in \mathbb{N}_0$
- but not $\sigma_0 \xrightarrow{\{go\}} \sigma_1 \xrightarrow{\{go,stop\}} \sigma_2$
  $\sigma_0 \not\models b \wedge off$

Isn't There a Bell Ringing...?

Isn't There a Bell Ringing...?

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
- $\alpha_i \in \mathcal{A}$, $i \in \mathbb{N}_0$, is called **action** (or **event**).

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

Isn't There a Bell Ringing...?

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
- $\alpha_i \in \mathcal{A}$, $i \in \mathbb{N}_0$, is called **action** (or **event**).

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

$\rightarrow$ a decision table T is **software** (**surprise, surprise**!?).

And Where's The Table?

And Where's The Table?

Decision tables can be written in tabular form:

T: room ventilation		r_1	r_2
b	button pressed?	×	×
off	ventilation off?	×	-
on	ventilation on?	-	×
go	start ventilation	×	-
$stop$	stop ventilation	-	×

And Where's The Table?

Decision tables can be written in tabular form:

T: room ventilation		r_1	r_2
b	button pressed?	×	×
off	ventilation off?	×	-
on	ventilation on?	-	×
go	start ventilation	×	-
$stop$	stop ventilation	-	×

From the table to the rules:

- $r_1 = b \wedge on \wedge \neg off \rightarrow \{go\}$
- $r_2 = b \wedge \neg on \wedge off \rightarrow \{stop\}$

And Where's The Table?

Decision tables can be written in tabular form:

T: room ventilation		r_1	r_2	r_3
b	button pressed?	$\times$	$\times$	$*$
off	ventilation off?	$\times$	-	-
on	ventilation on?	-	$\times$	-
go	start ventilation	$\times$	-	-
$stop$	stop ventilation	-	$\times$	-

From the table to the rules:

- $r_1 = b \wedge on \wedge \neg off \rightarrow \{go\}$
- $r_2 = b \wedge \neg on \wedge off \rightarrow \{stop\}$

And Where's The Table?

Decision tables can be written in tabular form:

T: room ventilation		r_1	r_2	r_3
b	button pressed?	$\times$	$\times$	$*$
off	ventilation off?	$\times$	-	-
on	ventilation on?	-	$\times$	-
go	start ventilation	$\times$	-	-
$stop$	stop ventilation	-	$\times$	-

From the table to the rules:

- $r_1 = b \wedge on \wedge \neg off \rightarrow \{go\}$
- $r_2 = b \wedge \neg on \wedge off \rightarrow \{stop\}$
- $r_3 = \neg on \wedge \neg off \rightarrow \emptyset$

Decision Tables vs. Rules In General

T: decision table		r_1	$\cdots$	r_n
c_1	description condition 1	$v_{1,1}$	$\cdots$	$v_{1,n}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
c_m	description condition m	$v_{m,1}$	$\cdots$	$v_{m,n}$
a_1	description action 1	$w_{1,1}$	$\cdots$	$w_{1,n}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
a_k	description action k	$w_{k,1}$	$\cdots$	$w_{k,n}$

$v_{i,j} \in \{-, \times, *\}$, $w_{i,j} \in \{-, \times\}$

Decision Tables vs. Rules In General

T: decision table		r_1	$\cdots$	r_n
c_1	description condition 1	$v_{1,1}$	$\cdots$	$v_{1,n}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
c_m	description condition m	$v_{m,1}$	$\cdots$	$v_{m,n}$
a_1	description action 1	$w_{1,1}$	$\cdots$	$w_{1,n}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
a_k	description action k	$w_{k,1}$	$\cdots$	$w_{k,n}$

$v_{i,j} \in \{-, \times, *\}$, $w_{i,j} \in \{-, \times\}$

- $C = \{c_1, \ldots, c_m\}$, $A = \{a_1, \ldots, a_k\}$
- $r_i = F(v_{1,i}, c_i) \wedge \cdots \wedge F(v_{m,i}, c_m) \rightarrow \{a_j \mid w_{j,i} = \times\}\}$, $\quad F(v, c) = \begin{cases} c & \text{, if } v = \times \\ \neg c & \text{, if } v = - \\ \mathit{true} & \text{, if } v = * \end{cases}$

  Recall: $\bigwedge_{1 \leq j \leq m} F(v_{j,i}, c_i) = \mathit{true}$ by definition.
- $T = \{r_1, \ldots, r_n\}$
- From rules to table: use disjunctive normal form of φ.

Decision Tables as Business Rules

$T1$: cash a cheque		r_1	r_2	else
c_1	credit limit exceeded?	×	×	
c_2	payment history ok?	×	-	
c_3	overdraft < 500 €?	-	$*$	
a_1	cash cheque	×	-	×
a_2	do not cash cheque	-	×	-
a_3	offer new conditions	×	-	-

(Balzert, 2009)

Decision Tables as Business Rules

$T1$: cash a cheque		r_1	r_2	else
c_1	credit limit exceeded?	×	×	
c_2	payment history ok?	×	-	
c_3	overdraft < 500 €?	-	$*$	
a_1	cash cheque	×	-	×
a_2	do not cash cheque	-	×	-
a_3	offer new conditions	×	-	-

(Balzert, 2009)

- One customer session at the bank:

$$\sigma \xrightarrow{\{cc, onc\}} \sigma'$$

if $\sigma \models c_1 \wedge c_2 \wedge \neg c_3$.

- clerk checks database state σ,
- database says: credit limit exceeded over 500 €, but payment history ok,
- clerk cashes cheque but offers new conditions.

Decision Tables as Software Specification

Another Bell...

- T can be viewed as a **software specification** by setting

$$[\![T]\!]_{spec} := \{S \mid [\![S]\!] \subseteq [\![T]\!]_{interleave}\}.$$

  in words:

  - Any software S, **whose behaviour is a subset** of T's behaviour, satisfies the specification T.

    In particular: S need not have all computation paths of T.

  - The computation paths of T are **all allowed** for the final product S; what is not a computation path of T is **forbidden** for the final product.

- **The refinement view**: Any software S' which **is a refinement** of a software $S \in [\![T]\!]_{spec}$ satisfies the specification T.

Basic Requirements Quality Criteria for DTs

Quality Criteria for DTs: Uselessness and Vacuity

Definition. [*Uselessness and Vacuity*] Let T be a decision table.

- A rule $r = \varphi \to \alpha$ is called **vacuous** (wrt. states Σ) if and only if there is no state $\sigma \in \Sigma$ such that

$$\sigma \models \varphi.$$

- A rule $r = \varphi \to \alpha$ is called **useless** (or: **redundant**) if and only if there is another rule r' whose premise φ' is subsumed by φ and whose effect is the same as r's, i.e. if

$$\exists\, r' = \varphi' \to \alpha', r' \neq r \bullet \varphi \implies \varphi' \wedge \alpha = \alpha'.$$

  r' is called **subsumed** by r.

Uselessness: Example

Example:

- $(c \wedge \neg c) \rightarrow \alpha$ is **vacuous**.

  Proposition: any rule with insatisfiable premise is vacuous.

Uselessness: Example

Example:

- $(c \wedge \neg c) \to \alpha$ is **vacuous**.

  Proposition: any rule with insatisfiable premise is vacuous.

- Let $\Sigma = \{\{c \mapsto 0\}\}$ — there's only one state $\sigma \in \Sigma$, and c not satisfied in σ.

  Then $c \to \alpha$ is **vacuous**.

Uselessness: Example

Example:

- $(c \wedge \neg c) \to \alpha$ is **vacuous**.

  Proposition: any rule with insatisfiable premise is vacuous.

- Let $\Sigma = \{\{c \mapsto 0\}\}$ — there's only one state $\sigma \in \Sigma$, and c not satisfied in σ.

  Then $c \to \alpha$ is **vacuous**.

-

T: room ventilation		r_1	r_2	r_3	r_4
b	button pressed?	$\times$	$\times$	-	-
off	ventilation off?	$\times$	-	$*$	-
on	ventilation on?	-	$\times$	$*$	-
go	start ventilation	$\times$	-	-	-
$stop$	stop ventilation	-	$\times$	-	-

  - r_4 is **useless** — it is subsumed by r_3.
  - r_3 is **not subsumed** by r_4!

Uselessness: Example

Example:

- $(c \wedge \neg c) \to \alpha$ is **vacuous**.

  Proposition: any rule with insatisfiable premise is vacuous.

- Let $\Sigma = \{\{c \mapsto 0\}\}$ — there's only one state $\sigma \in \Sigma$, and c not satisfied in σ.

  Then $c \to \alpha$ is **vacuous**.

-

T: room ventilation		r_1	r_2	r_3	r_4
b	button pressed?	×	×	-	-
off	ventilation off?	×	-	$*$	-
on	ventilation on?	-	×	$*$	-
go	start ventilation	×	-	-	-
$stop$	stop ventilation	-	×	-	-

  - r_4 is **useless** — it is subsumed by r_3.
  - r_3 is **not subsumed** by r_4!

- **Proposition**: if rule r is given in form of a table and if Σ is equivalent to $C \to \mathbb{B}$, then r is not vacuous (yet it may be subsumed by another rule).

Uselessness: Consequences

- **Doesn't hurt** wrt. the final product:

  The decision table T with useless rules has the same computation paths as the one with useless rules removed.

- **But**
  - Decision tables with useless rules are unnecessarily hard to work with (read, maintain, ...).
  - **May make communication with customer harder!**

Quality Criteria for DTs: Completeness

Definition. [*Completeness*] A decision table T is called **complete** if and only if the disjunction of all rules' premises is a tautology, i.e. if

$$\models \bigvee_{\varphi \to \alpha \in T} \varphi.$$

Completeness: Example

T: room ventilation		r_1	r_2
b	button pressed?	$\times$	$\times$
off	ventilation off?	$\times$	-
on	ventilation on?	-	$\times$
go	start ventilation	$\times$	-
$stop$	stop ventilation	-	$\times$

Completeness: Example

T: room ventilation		r_1	r_2
b	button pressed?	$\times$	$\times$
off	ventilation off?	$\times$	-
on	ventilation on?	-	$\times$
go	start ventilation	$\times$	-
$stop$	stop ventilation	-	$\times$

- is **not complete**: there is no rule, e.g., for the case $\neg b \wedge on \wedge \neg off$.

Completeness: Example

T: room ventilation		r_1	r_2
b	button pressed?	×	×
off	ventilation off?	×	-
on	ventilation on?	-	×
go	start ventilation	×	-
$stop$	stop ventilation	-	×

- is **not complete**: there is no rule, e.g., for the case $\neg b \wedge on \wedge \neg off$.

T: room ventilation		r_1	r_2	r_3	r_4	r_5
b	button pressed?	×	×	-	$*$	$*$
off	ventilation off?	×	-	$*$	×	-
on	ventilation on?	-	×	$*$	×	-
go	start ventilation	×	-	-	-	-
$stop$	stop ventilation	-	×	-	-	-

- is complete.

Completeness: Example

T: room ventilation		r_1	r_2
b	button pressed?	$\times$	$\times$
off	ventilation off?	$\times$	-
on	ventilation on?	-	$\times$
go	start ventilation	$\times$	-
$stop$	stop ventilation	-	$\times$

- is **not complete**: there is no rule, e.g., for the case $\neg b \wedge on \wedge \neg off$.

T: room ventilation		r_1	r_2	else
b	button pressed?	$\times$	$\times$	
off	ventilation off?	$\times$	-	
on	ventilation on?	-	$\times$	
go	start ventilation	$\times$	-	-
$stop$	stop ventilation	-	$\times$	-

- is complete.

Incompleteness: Consequences

- An incomplete decision table may allow too little behaviour (it forbids too much)!

- This very incomplete decision table:

T: room ventilation		r_1
b	button pressed?	$\times$
off	ventilation off?	$\times$
on	ventilation on?	-
go	start ventilation	$\times$
$stop$	stop ventilation	-

- forbids all actions in case $b \wedge \neg on \wedge off$ is satisfied.
- **May not be the intention of the customer!**

Quality Criteria for DTs: Determinism

Definition. [*Determinism*] A decision table T is called **deterministic** if and only if the premises of all rules are pairwise disjoint, i.e. if

$$\forall\, \varphi_i \to \alpha_i, \varphi_j \to \alpha_j, i \neq j \bullet \models \neg(\varphi_i \wedge \varphi_j).$$

Otherwise, T is called **non-deterministic**.

Determinism: Example

T: room ventilation		r_1	r_2	r_3
b	button pressed?	$\times$	$\times$	-
go	start ventilation	$\times$	-	-
$stop$	stop ventilation	-	$\times$	-

- is **non-determistic**:

Determinism: Example

T: room ventilation		r_1	r_2	r_3
b	button pressed?	$\times$	$\times$	-
go	start ventilation	$\times$	-	-
$stop$	stop ventilation	-	$\times$	-

- is **non-determistic**: In a state σ with $\sigma \models b$, both rules are enabled.

Determinism: Example

T: room ventilation		r_1	r_2	r_3
b	button pressed?	$\times$	$\times$	-
go	start ventilation	$\times$	-	-
$stop$	stop ventilation	-	$\times$	-

- is **non-determistic**: In a state σ with $\sigma \models b$, both rules are enabled.
- Is non-determinism **a bad thing** in general?

Determinism: Example

T: room ventilation		r_1	r_2	r_3
b	button pressed?	$\times$	$\times$	-
go	start ventilation	$\times$	-	-
$stop$	stop ventilation	-	$\times$	-

- is **non-determistic**: In a state σ with $\sigma \models b$, both rules are enabled.
- Is non-determinism **a bad thing** in general?
  - **Just the opposite**: one of the most powerful **modelling tools** we have.
  - Read table T as:
    - **under certain conditions** (which I will specify later) **the button** may switch the ventilation **on**, and
    - **under certain conditions** (which I will specify later) **the button** may switch the ventilation **off**.
  - This is quite some less chaos than full chaos!
  - We can already analyse the specification, e.g., we state that we do not (under any condition) want to see on and off executed together.

Non-determism: Consequences

- **Good**:
  - A non-determistic decision table leaves **more choices** (more freedom) to the developer.

- **Bad**:
  - A non-determistic decision table leaves **more choices** to the developer; even the choice to create a **non-deterministic final product**.

    (Input-)**Deterministic final products**, i.e. "same data in, same data out", are easier to deal with and **are usually desired**.

- **Another view**:
  deterministic decision tables can be **implemented** with deterministic programming languages.

Implementing Decision Tables

T: room ventilation		r_1	r_2	else
b	button pressed?	×	×	
off	ventilation off?	×	-	
on	ventilation on?	-	×	
go	start ventilation	×	-	-
$stop$	stop ventilation	-	×	-

```
1  int b, on, off;
2
3  extern void go();  extern void stop();
4
5  void (*effect)() = 0;
6
7  void dt() {
8
9    read_b_on_off(); // read
10
11   // compute
12   //
13   if (b && off) effect = go;
14
15   if (b && on) effect = stop;
16
17   execute_effect(); // write
18 }
```

More Requirements Quality Criteria for DTs

Consistency of Rules

T: room ventilation		r_1	r_2	else
b	button pressed?	×	×	
off	ventilation off?	×	-	
on	ventilation on?	-	×	
go	start ventilation	×	-	-
$stop$	stop ventilation	-	×	-

- Conditions and actions are **abstract entities** without inherent connection to the **real world**
- Yet we **want to use** decision tables to model/represent requirements on the behaviour of software systems, which are **used** in the real world.
- When modelling **real-world aspects** by conditions and actions, we should also model **relations between actions/conditions** in the real-world ($\rightarrow$ domain model).

Consistency of Rules

T: room ventilation		r_1	r_2	else
b	button pressed?	×	×	
off	ventilation off?	×	-	
on	ventilation on?	-	×	
go	start ventilation	×	-	-
$stop$	stop ventilation	-	×	-

- Conditions and actions are **abstract entities** without inherent connection to the **real world**
- Yet we **want to use** decision tables to model/represent requirements on the behaviour of software systems, which are **used** in the real world.
- When modelling **real-world aspects** by conditions and actions, we should also model **relations between actions/conditions** in the real-world ($\rightarrow$ domain model).

Example:

- if on models "room ventilation is on", and
- if off models "room ventilation is off",
- then in the abstract setting, $\sigma \models on \wedge off$ is still possible.

  T "doesn't know" that on and off model opposites in the real-world.

Conflicting Conditions and Actions, Consistency

- A **conflict axiom** for conditions C is a formula $\varphi_{confl} \in \Phi(C)$.

Definition. [*Vacuitiy wrt. Conflict Axiom*] A rule $r = \varphi \to \alpha \in T$ over C and A is called **vacuous wrt. conflict axiom** $\varphi_{confl} \in \Phi(C)$ if and only if the premise implies the conflict axiom, i.e. if $\models (\varphi \implies \varphi_{confl})$.

Conflicting Conditions and Actions, Consistency

- A **conflict axiom** for conditions C is a formula $\varphi_{confl} \in \Phi(C)$.

Definition. [*Vacuitiy wrt. Conflict Axiom*] A rule $r = \varphi \to \alpha \in T$ over C and A is called **vacuous wrt. conflict axiom** $\varphi_{confl} \in \Phi(C)$ if and only if the premise implies the conflict axiom, i.e. if $\models (\varphi \implies \varphi_{confl})$.

- A **conflict relation** on actions A is a **transitive** and **symmetric** relation $\lightning \subseteq (A \times A)$.

Definition. [*Consistency*] Let $r = \varphi \to \alpha \in T$ be a rule.

(i) r is called **consistent with conflict relation** $\lightning$ if and only if there are no conflicting actions in α, i.e. if $\nexists\, a_1, a_2 \in \alpha \bullet a_1 \lightning a_2$.

(ii) T is called **consistent** if and only if all rules $r \in T$ are consistent.

Example: Conflicting Conditions Actions

- Let $\varphi_{confl} = (on \wedge off) \wedge (\neg on \wedge \neg off)$.

  "on models an opposite of off, neither can both be satisfied nor bot non-satisfied"

- Let $\lightning$ be the transitive, symmetric closure of $stop \lightning go$.

  "actions $stop$ and go are not supposed to be executed at the same time"

- Then

T: room ventilation		r_1	r_2
b	button pressed?	×	×
off	ventilation off?	×	-
on	ventilation on?	×	×
go	start ventilation	×	×
$stop$	stop ventilation	-	×

  - r_1 is vacuous wrt. φ_{confl},
  - r_2 is inconsistent with $\lightning$.

Conflicting Conditions Actions: Consequences

- **Vacuity** wrt. φ_{confl}:

  Same as with uselessness and general vacuity, **doesn't hurt** but

  May make communication with customer harder!

  Implementing vacuous rules is a waste of effort!

- **Consistency**:

  A decision table with **non-useless/vacuous** but **inconsistent** rules **cannot be implemented!**

  Detecting an inconsistency only late during a project can incur significant cost!

  Inconsistencies (in particular in (multiple) decision tables, created and edited by multiple people, as well as in requirements in general) are **not always as obvious** as in the toy examples given here! (would be too easy...)

Other Semantics for Decision Tables

A Collecting Semantics for Decision Tables

- Let T be a decision table and $\pi = \sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots$ a state/event sequence.
- **Recall**: π is a **computation path** of T (in the interleaving semantics) if

$$\forall\, i \in \mathbb{N}_0\ \exists\, r = \varphi \to \alpha \in T \bullet \sigma_i \models \varphi \wedge \alpha_{i+1} = \alpha.$$

- π is a **computation path** of T (in the **collecting semantics**) if and only if

$$\forall\, i \in \mathbb{N}_0\ \exists\, r = \varphi \to \alpha \in T \bullet \sigma_i \models \varphi \wedge \alpha_{i+1} = \bigcup_{\substack{\varphi' \to \alpha' \in T, \\ \sigma_i \models \varphi'}} \alpha'.$$

That is, **all** rules which are **enabled** in σ_i "fire" simultaneously,
the joint effect is the union of the effects.

- **Advantage**:
  - separation of concerns, multiple (smaller) tables may contribute to a transition,
  - no non-determinism between rules: all enabled ones "fire".
- **Disadvantages**: conflicts much less obvious.

An Update Semantics for Decision Tables

- By now, we didn't talk about the **effect of actions** from A on states.
  Actions are **uninterpreted**.
- **Recall** the **"cash cheque"** example:

  Here it makes sense, because the next state seen by the bank clerk may be the result of many database updates by other bank clerks.

We can also define a semantics with action effects:

- Let A be a set of actions and Σ a set of states.
- Let $[\![\cdot]\!]_{acteff} : A \times \Sigma \to \Sigma$ which assigns to each pair of (a, σ) of action and state a new state σ'. σ' is called **the result** of applying a to σ.
- **Example**: on $\Sigma = \{b, on, off\}$, we could define

$$[\![go]\!]_{acteff}(\sigma) = \sigma|_{\{b\}} \cup \{on \mapsto 1, off \mapsto 0\}$$

$$[\![stop]\!]_{acteff}(\sigma) = \sigma|_{\{b\}} \cup \{on \mapsto 0, off \mapsto 1\}$$

- The interleaving semantics with action effects is then

$$\forall\, i \in \mathbb{N}_0\ \exists\, r = \varphi \to \alpha \in T \bullet \sigma_i \models \varphi \wedge \alpha_{i+1} = \alpha \wedge \sigma_{i+1} = [\![\alpha]\!]_{acteff}(\sigma).$$

An Update Semantics for Decision Tables Cont'd

- In addition, we may want to constrain initial states, i.e. give a set $\Sigma_{ini} \subseteq \Sigma$.
- Computation paths of T (over Σ) are then required to have $\sigma_0 \in \Sigma_{ini}$.

- **Example**: decision table

T: room ventilation		r_1
off	ventilation off?	×
on	ventilation on?	-
go	start ventilation	×
stop	stop ventilation	-

with $\Sigma_{ini} = \{\{on \mapsto 0, off \mapsto 1\}\}$ has only one computation path, namely

$$\sigma_0 \xrightarrow{off} \sigma_1$$

with $\sigma_1 = \{on \mapsto 0, off \mapsto 1\}$.

- We can say T **terminates**.
- This gives rise to **another notion of vacuity**: $r = (on \wedge off) \rightarrow \alpha$ is never enabled, because no state satisfying the premise is ever reached (even with conflict axiom $\varphi_{confl} = \mathit{false}$).

Distinguishing Controlled and Uncontrolled Conditions

- For some systems, we can distinguish **inputs** and **outputs**.
- **In terms of decision tables**:
  - C is partitioned into **controlled conditions** C_c and **uncontrolled conditions** C_u, i.e. $C = C_c \dot\cup C_u$.
  - actions only affect controlled conditions.
- **Example**:
  - $C_c = \{on, off\}$ (only the software switches the ventilation on or off),
  - $C_u = \{b\}$ (the button is not controlled by the software, but by the environment, by a user external to the computer system)
- One more quality criterion, **another notion of completeness**:

  We want the specification to be able to deal with all possible sequences of inputs, i.e. we require

$$[\![T]\!]|_{C_u} = (\Sigma|_{C_u})^\omega \quad (\text{ or } (\Sigma|_{C_u})^*).$$

Controlled and Uncontrolled Conditions: Example

- **Example**:

T: room ventilation		r_1	r_2
b	button pressed?	×	-
off	ventilation off?	×	-
on	ventilation on?	-	×
go	start ventilation	×	-
$stop$	stop ventilation	-	×

  - is not **input sequence complete**:

    There is no rule enabled if the button is pressed when the ventilation is off.

  - **Note**: it's not that pressing the button such a state **has no effect**, but the system **stops to work**; it "gets stuck" in that state.

    (Because in order to take a transition, we need to have at least one enabled rule.)

Decision Tables: Discussion

User Stories

Scenarios

Live Sequence Charts

Use Case Diagrams

Wrap-Up

Software vs. Systems Engineering

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.

Balzert, H. (2009). *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*. Spektrum, 3rd edition.

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

DIN (2009). *Projektmanagement; Projektmanagementsysteme*. DIN 69901-5.

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.