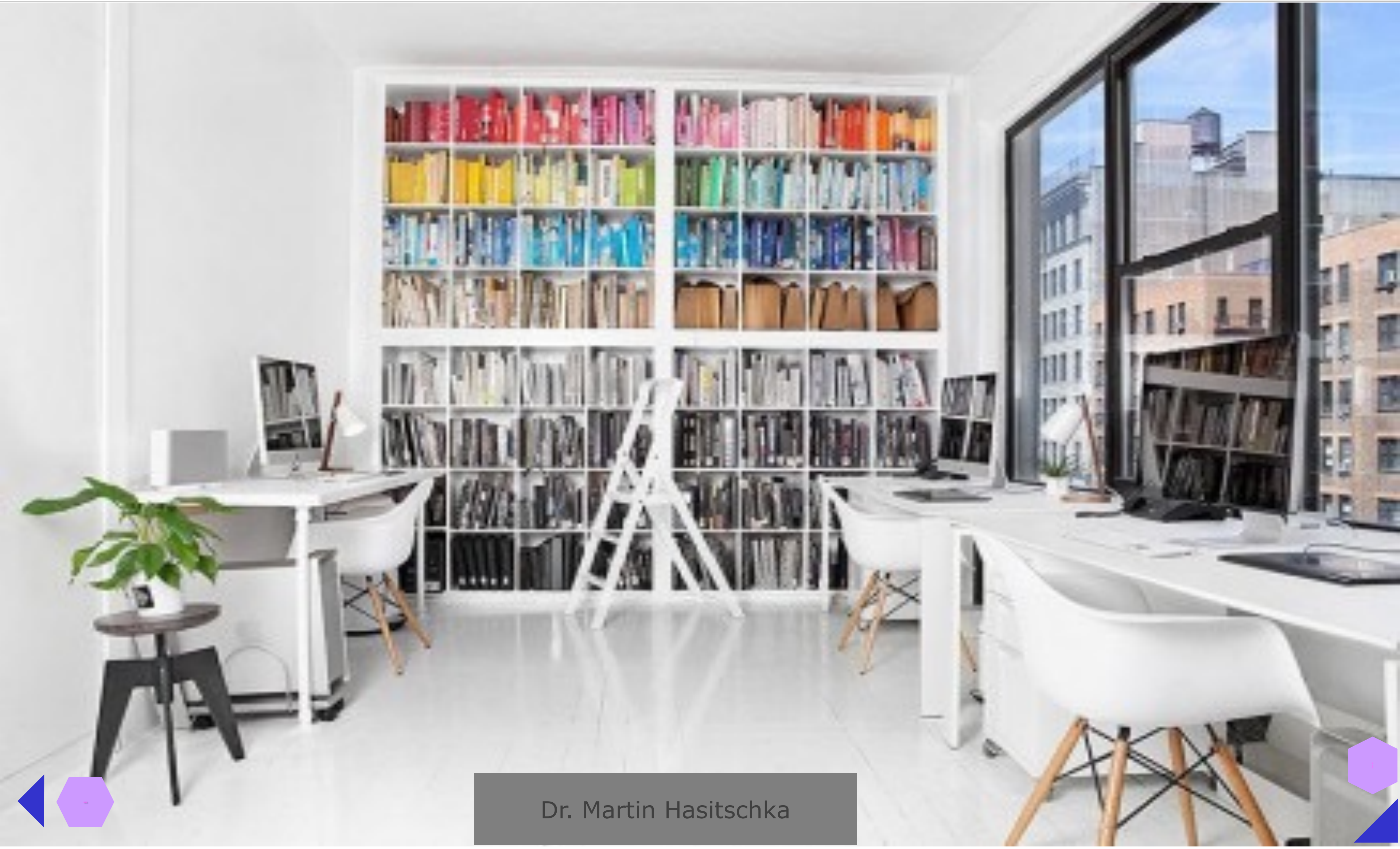
Requirements Structuring



Dr. Martin Hasitschka

Dashboard

SE&MB+G
pre-upload SE&M
with optional

SE4AIB+G
pre-upload SE4AI
with optional

SE&MB
pre-upload SE&M
without optional

SE4AIB
pre-upload SE4AI
without optional

SE&MI
presentation SE&M

SE4AI**B+**I
presentation SE4AI

SE&MB+G+O
post-upload SE&M

SE4AIB+G+O
post-upload SE4AI

Competence Map:
At the End of this Course you will ...

1. Top 10 Problems of RE

...know the top 10 problems of RE based on a real-world example

2. Requirements Elicitation

... know how to elicit
basic requirements
performance requirements
excitement requirements
superfluous requirements

3. Requirements Structuring

... be able to structure

requirements documents

individual requirements

requirement backlogs

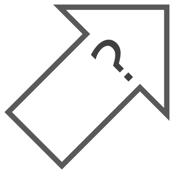
4. Requirements Documentation

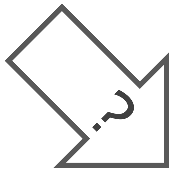
... be able to write high-quality natural language requirements both for humans and AI systems

... be able to choose the right diagrams for requirements

Content

results of requirements elicitation





Content

Structuring...

The Whole Body of Requirements

Individual Requirements

Content

Structuring…

The Whole Body of Requirements

Individual Requirements

System Requirements and Requirements Specifications

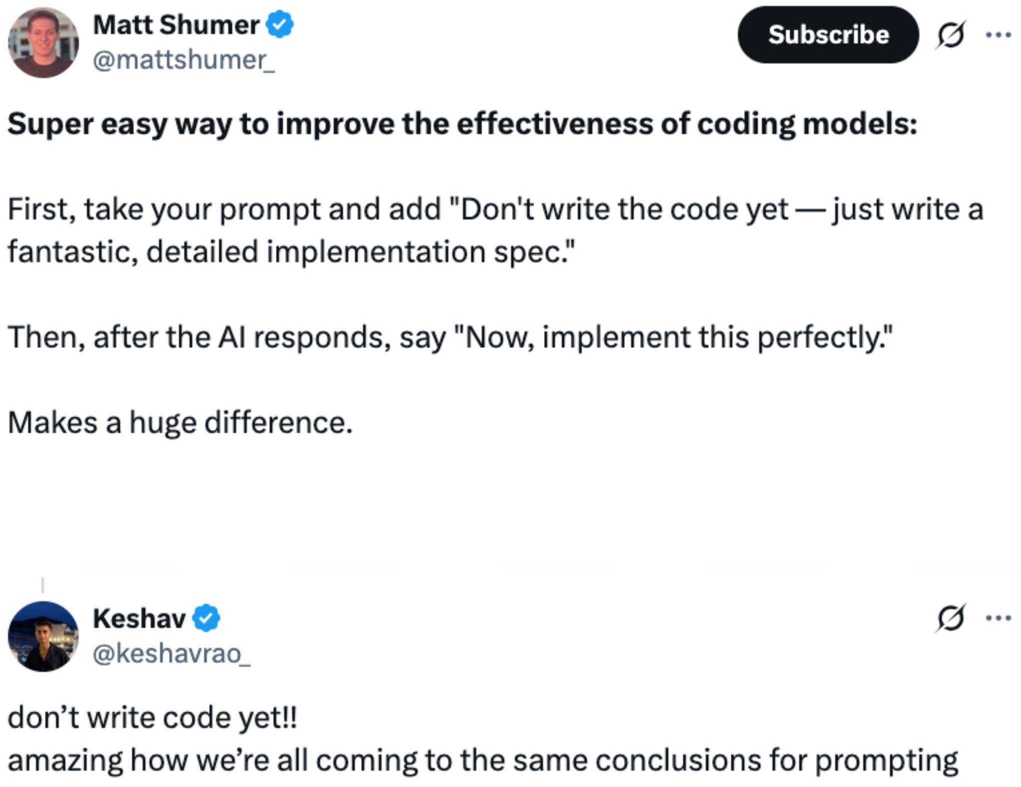
Very often, you have **system requirements**, worded in the **language of the application domain**.

Example: SYS1: *„The system prevents the user from entering a ski slope that is too difficult for him/her"*.

The system requirements are broken down into **requirements specifications**, worded in the **language of the product.**

Examples:
SPEC1: *„Using the speed sensor, the system collects data about near accidents of the user: A rapid drop in speed (-20 km/h in < 1 sec) followed by a standstill that lasts at least 30 minutes is counted as a near accident."*
SPEC2: *„Using the GPS sensor, the system collects data about the total number of skiing kilometers."*
SPEC3: *„The system computes the skill class. Altogether there are five skill classes. The user is in skill class 1 if there is one near accident per at most 1000 kilometers, class 2 if … The best skill class is 5."*
SPEC4: *„The system gets the current difficulty of the slope via the API-calls like https://api.skiresort.com/ski-slope/difficulty?lat=48.8566&long=2.3522."*
SPEC5: *„The system displays the warning „SLOPE TOO DIFFICULT" if the user enters a slope that is too difficult compared to the skill level of the user, i.e. more than one full degree class their skill level."*

The Role of Requirements Specifications



Matt Shumer @mattshumer_

Super easy way to improve the effectiveness of coding models:

First, take your prompt and add "Don't write the code yet — just write a fantastic, detailed implementation spec."

Then, after the AI responds, say "Now, implement this perfectly."

Makes a huge difference.

Keshav @keshavrao_

don't write code yet!!
amazing how we're all coming to the same conclusions for prompting

Source: https://x.com/mattshumer_/status/1894789818925609124?s=46&t=liJ6-bCACnb-WfoI5TZ2RA , 2025

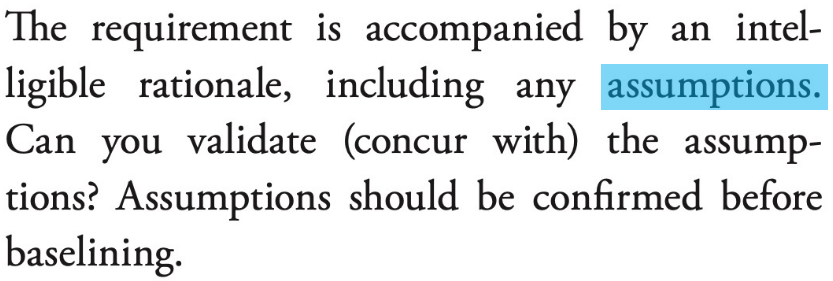
Assumptions

The Requirements Specification typically rests on **assumptions**.

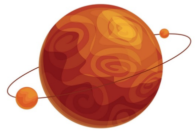
Examples:
AS1: *„The speed sensor is working."*
AS2: *„The GPS sensor is working."*
AS3: *„A near accident can be detected by a rapid drop in speed (-20 km/h in < 1 sec) followed by a standstill that lasts at least 30 minutes."*
AS4: *„User skill level data is always available".*
AS5: *„The API exists and is working."*
AS6: *„The API returns a number between 1 and 5."*
AS7: *„For the API, 5 is the highest skill level (not 1)."*
AS8: *„The data returned by the API is up to date."*
AS9: *„The display is warm enough to show warning messages."*
AS10: *„There is enough power in the battery to operate the display."*
AS11: *„Users can handle slopes that are up to one skill level above their abilities."*

Note the following connections: **If some of the assumptions are wrong, then the requirements specification does not fulfil the system requirements**.

Assumptions@NASA



The requirement is accompanied by an intelligible rationale, including any assumptions. Can you validate (concur with) the assumptions? Assumptions should be confirmed before baselining.



AS2: Every day, each Mars moon rises and sets at certain points in time. Both points in time are expressed as *Mars-timestamps*, together forming a *Mars-interval*. Deimos, for instance, could rise at 4:97 and set at 12:02. That day, Phobos could rise at 24:44 and set at 7:50 the next day.

AS4: It may be assumed that a moon rises and sets exactly in the middle of a Mars-minute.

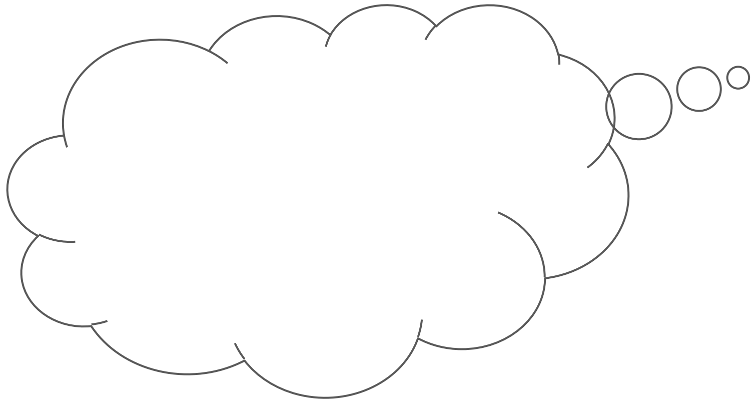
Source: https://www.nasa.gov/wp-content/uploads/2018/09/nasa_systems_engineering_handbook_0.pdf, 2025

Exercise: System Requirements and Requirements Specifications

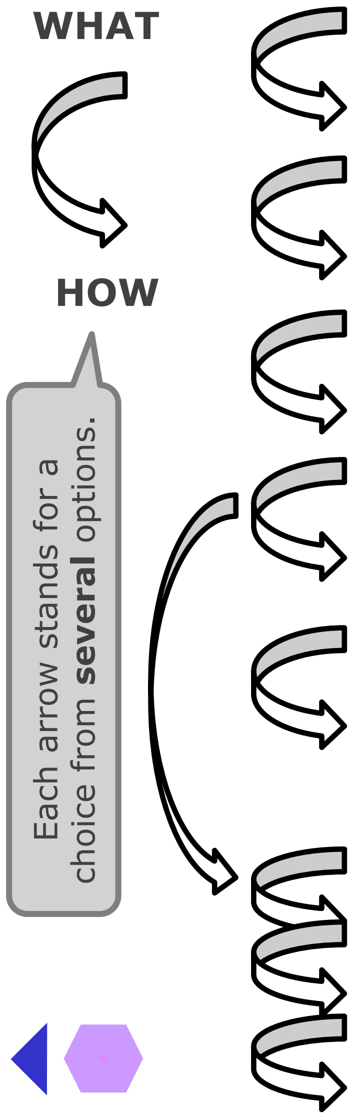
You have the following system requirement:

SYS7: *„The system automatically notifies first responders if the user is buried by an avalanche.*"

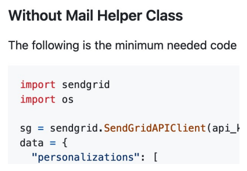
Break it down into a **requirements specification.** Also, document your **assumptions.**



Where are the Boundaries Between System Requirements, Specification and Design?

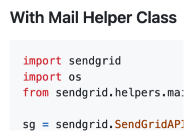


1. The software should contribute to making the library customers happy.
2. The software should offer as many books as possible for lending so that the customers happier.
3. The software should make sure less books are overdue so that more books are offered for lending.
4. The software should send reminder-emails so that less books are overdue.
5. Each reminder-email should contain the title of the book and the due date.
6. The due date should be displayed in the format dd-mm-yyyy.
7. The mails should be sent via a REST-API.
8. The mails should be sent via the sendgrid-API.
9. The mails should be sent using a mail helper class.
10. The mail helper class should be called Mhlp.


Without Mail Helper Class

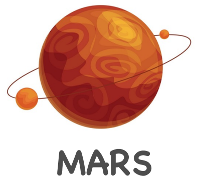
The following is the minimum needed code

```
import sendgrid
import os

sg = sendgrid.SendGridAPIClient(api_k
data = {
  "personalizations": [
```


With Mail Helper Class

```
import sendgrid
import os
from sendgrid.helpers.ma

sg = sendgrid.SendGridAPI
```

The „Flowdown of Requirements“



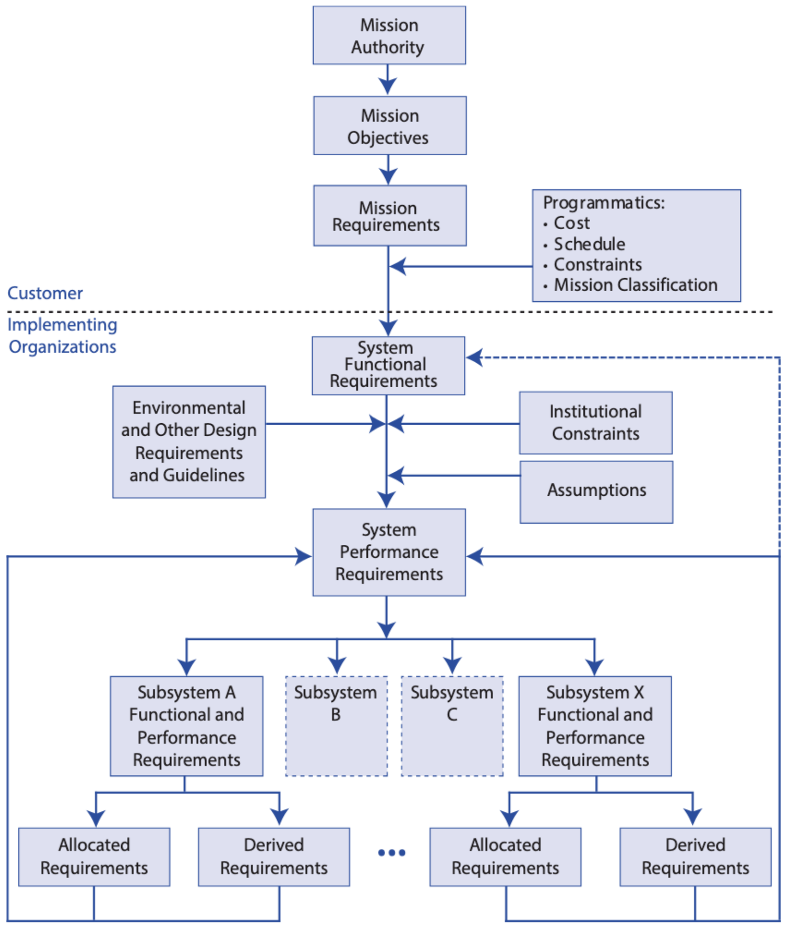

Mission Authority
Mission Objectives
Mission Requirements
Programmatics:
• Cost
• Schedule
• Constraints
• Mission Classification
Customer
Implementing Organizations
System Functional Requirements
Environmental and Other Design Requirements and Guidelines
Institutional Constraints
Assumptions
System Performance Requirements
Subsystem A Functional and Performance Requirements
Subsystem B
Subsystem C
Subsystem X Functional and Performance Requirements
Allocated Requirements
Derived Requirements
...
Allocated Requirements
Derived Requirements

FIGURE 4.2-3 The Flowdown of Requirements

Source: https://www.nasa.gov/wp-content/uploads/2018/09/nasa_systems_engineering_handbook_0.pdf, 2025

Requirements Specifications and Machine Learning

You have the following system requirement:

SYS7: *„The system automatically notifies first responders if the user is buried by an avalanche."*

If you fail to write a good specification, you may still have data from goggles that were buried in avalanches (along with their wearer), much like flight recorders. You can learn from this data what combinations of sensor readings are typical of getting caught in an avalanche. So **machine learning can replace requirements specifications**:

*„ We use machine learning **precisely because we do not have such specifications**, typically because the problem is too complex to specify or the rules are simply unknown." (Christian Kästner)*

*Note that Andrej Karpathy calls software created this way **Software 2.0**.*
„Software 2.0 is eating Software 1.0"

Content

Structuring...

The Whole Body of Requirements

Requirements Documents

Requirements Backlogs

Individual Requirements

Are There Still Requirements **Documents**?

Many development teams don‵t get requirements shaped as text documents, but as a backlog in a tool.

There are other situations, though, in which requirements are shaped as (often rather long) text documents:

- If a software customer is looking for a software supplier, the customer typically writes the **Request for Proposal** (abbreviation: RfP, German: *Ausschreibung*) as a text document describing the requirements. It can be quite long and therefore needs careful structuring.

Are There Still Requirements **Documents**? / 2

- Conversely, if a software product supplier is looking for customers, the supplier will need some document that **describes the features of the product**. That document is a requirements document.

- If two parties agree on a fixed-price development project and sign a contract, that **contract in most cases contains a description of the software** to be developed. That description is a requirements document.

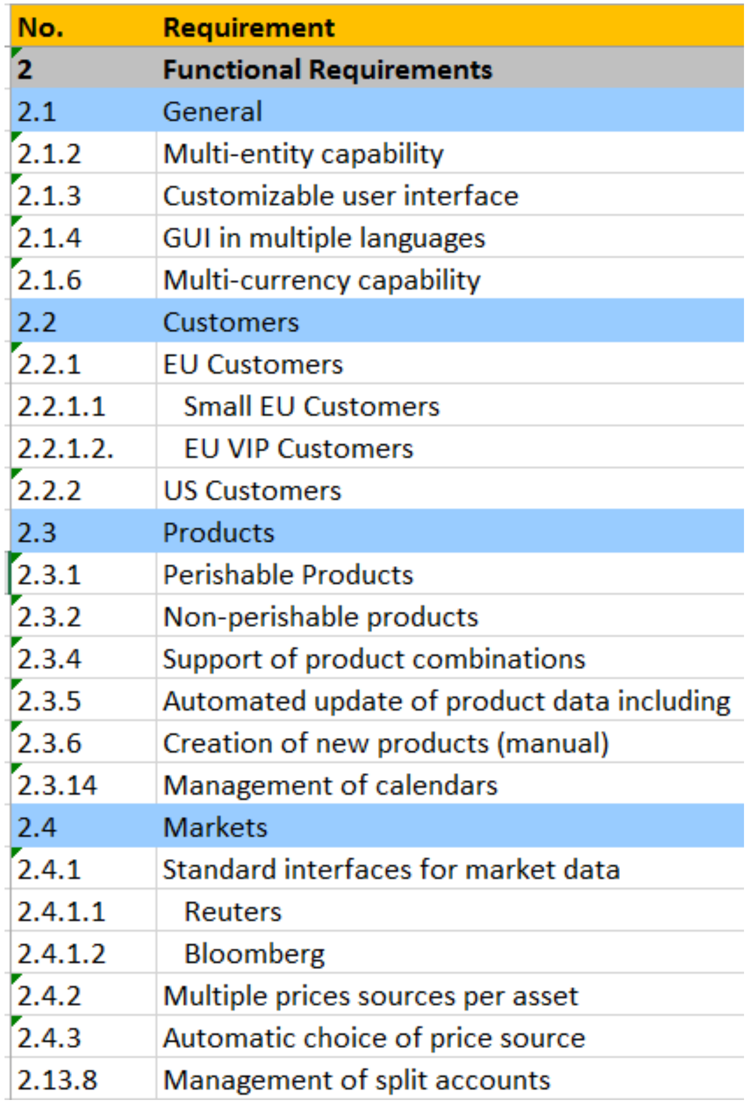
In many cases, you live in an intermediate world between perfect agility and total waterfall. You have a rough requirements document at the start of the project, you convert it into a backlog as one of your first tasks in the project and you refine the backlog over the course of the project.

The Top-10 RE Problems in Practice

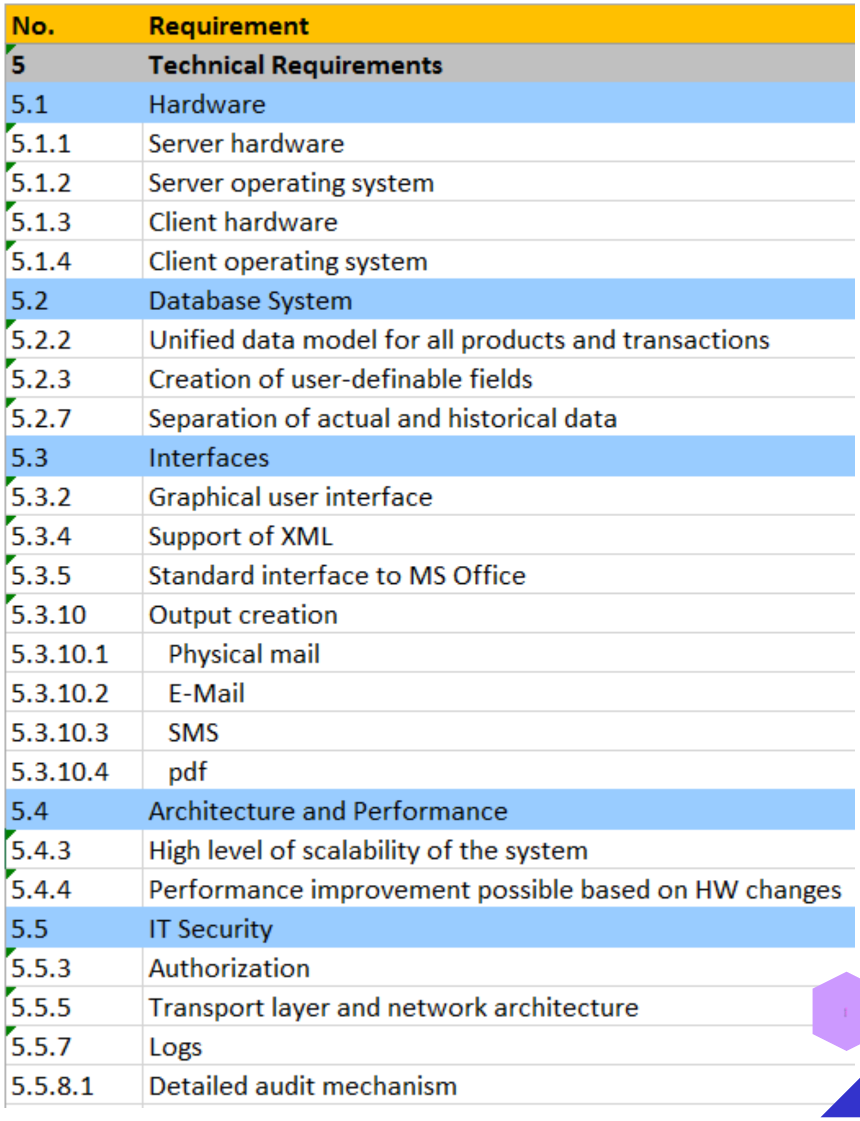
In a perfect world, we have **one** customer who provides us with high-quality requirements. In practice, however, we need to cope with questions like ...

1. How can we find **all** sources of requirements?
2. What can we do if the requirements sources disagree?
3. How can we elicit the requirements the customer has but doesn't tell us?
4. How can we find out if the customer **really** needs what she says?
5. What is the best prioritization method for requirements?
6. What is the best table of contents for a big requirements document?
7. What can we do to get **precise** natural language requirements?
8. What are the best diagram types for visualizing requirements?
9. What are the most important requirements attributes and relationships?
10. What can we do to keep the requirements up-to-date for years?

Requirements Document: Example (Banking)

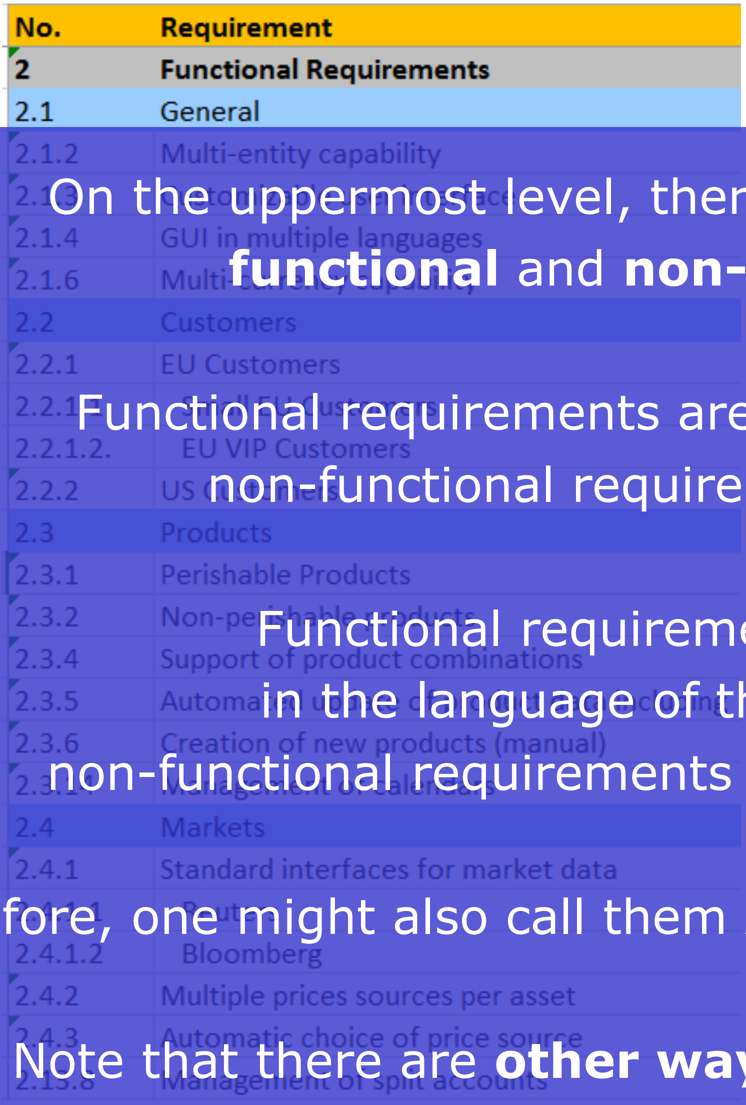


No.	Requirement
2	**Functional Requirements**
2.1	General
2.1.2	Multi-entity capability
2.1.3	Customizable user interface
2.1.4	GUI in multiple languages
2.1.6	Multi-currency capability
2.2	Customers
2.2.1	EU Customers
2.2.1.1	Small EU Customers
2.2.1.2.	EU VIP Customers
2.2.2	US Customers
2.3	Products
2.3.1	Perishable Products
2.3.2	Non-perishable products
2.3.4	Support of product combinations
2.3.5	Automated update of product data including
2.3.6	Creation of new products (manual)
2.3.14	Management of calendars
2.4	Markets
2.4.1	Standard interfaces for market data
2.4.1.1	Reuters
2.4.1.2	Bloomberg
2.4.2	Multiple prices sources per asset
2.4.3	Automatic choice of price source
2.13.8	Management of split accounts

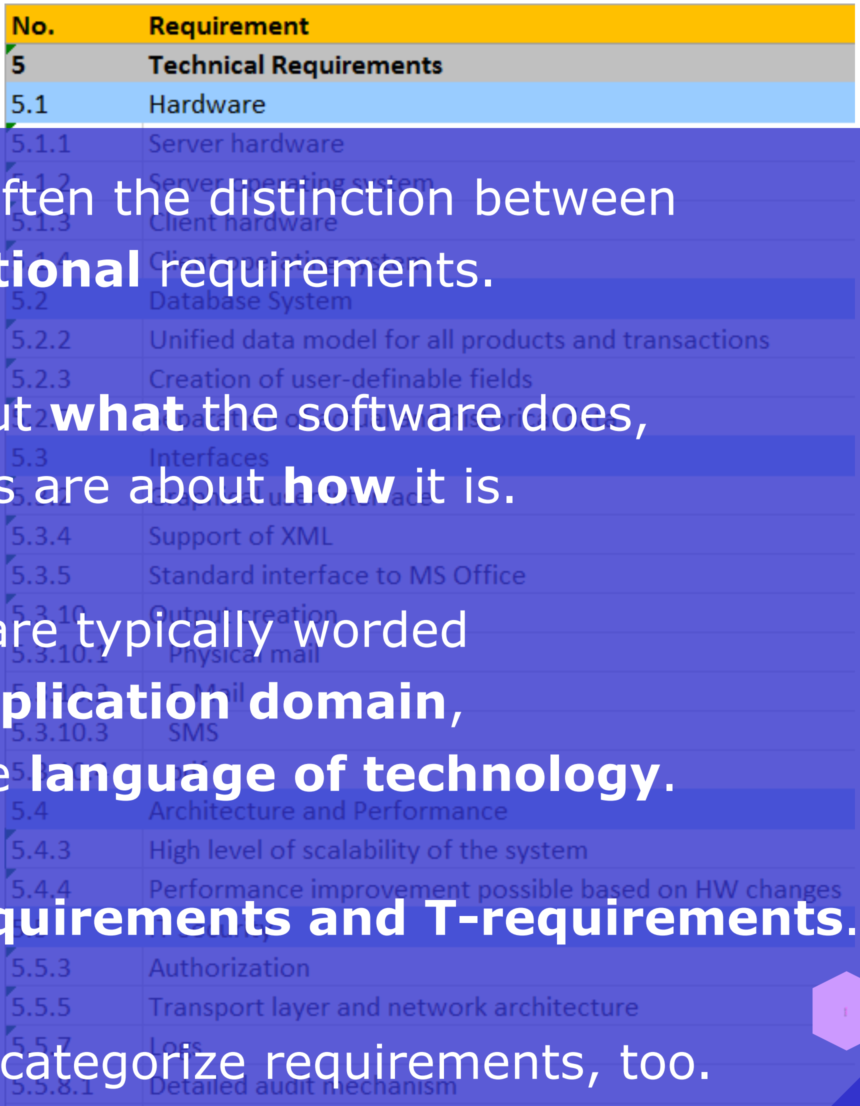


No.	Requirement
5	**Technical Requirements**
5.1	Hardware
5.1.1	Server hardware
5.1.2	Server operating system
5.1.3	Client hardware
5.1.4	Client operating system
5.2	Database System
5.2.2	Unified data model for all products and transactions
5.2.3	Creation of user-definable fields
5.2.7	Separation of actual and historical data
5.3	Interfaces
5.3.2	Graphical user interface
5.3.4	Support of XML
5.3.5	Standard interface to MS Office
5.3.10	Output creation
5.3.10.1	Physical mail
5.3.10.2	E-Mail
5.3.10.3	SMS
5.3.10.4	pdf
5.4	Architecture and Performance
5.4.3	High level of scalability of the system
5.4.4	Performance improvement possible based on HW changes
5.5	IT Security
5.5.3	Authorization
5.5.5	Transport layer and network architecture
5.5.7	Logs
5.5.8.1	Detailed audit mechanism

Requirements Document: Example (Banking)



No.	Requirement
2	Functional Requirements
2.1	General
2.1.2	Multi-entity capability
2.1.3	[illegible]
2.1.4	GUI in multiple languages
2.1.6	Multi-currency capability
2.2	Customers
2.2.1	EU Customers
2.2.1.1	Small EU Customers
2.2.1.2.	EU VIP Customers
2.2.2	US Customers
2.3	Products
2.3.1	Perishable Products
2.3.2	Non-perishable products
2.3.4	Support of product combinations
2.3.5	Automa[illegible]
2.3.6	Creation of new products (manual)
2.3.[illegible]	Management of calendars
2.4	Markets
2.4.1	Standard interfaces for market data
2.4.1.1	Reuters
2.4.1.2	Bloomberg
2.4.2	Multiple prices sources per asset
2.4.3	Automatic choice of price source
2.15.8	Management of split accounts



No.	Requirement
5	Technical Requirements
5.1	Hardware
5.1.1	Server hardware
5.1.2	Server operating system
5.1.3	Client hardware
5.1.4	Client operating system
5.2	Database System
5.2.2	Unified data model for all products and transactions
5.2.3	Creation of user-definable fields
5.2.[illegible]	[illegible]
5.3	Interfaces
5.3.[illegible]	[illegible]
5.3.4	Support of XML
5.3.5	Standard interface to MS Office
5.3.10	Output creation
5.3.10.1	Physical mail
5.3.10.2	E-Mail
5.3.10.3	SMS
[illegible]	[illegible]
5.4	Architecture and Performance
5.4.3	High level of scalability of the system
5.4.4	Performance improvement possible based on HW changes
5.5	Security
5.5.3	Authorization
5.5.5	Transport layer and network architecture
5.5.7	Logs
5.5.8.1	Detailed audit mechanism

On the uppermost level, there is often the distinction between **functional** and **non-functional** requirements.

Functional requirements are about **what** the software does, non-functional requirements are about **how** it is.

Functional requirements are typically worded in the language of the **application domain**, non-functional requirements in the **language of technology**.

Therefore, one might also call them **A-requirements and T-requirements**.

Note that there are **other ways** to categorize requirements, too.

Another Way to Categorize Requirements

As you know from testing, there is

- **positive** testing: Testing whether the system does what it should do.
- **negative** testing: Testing whether the system does not what it shouldn�much t do.

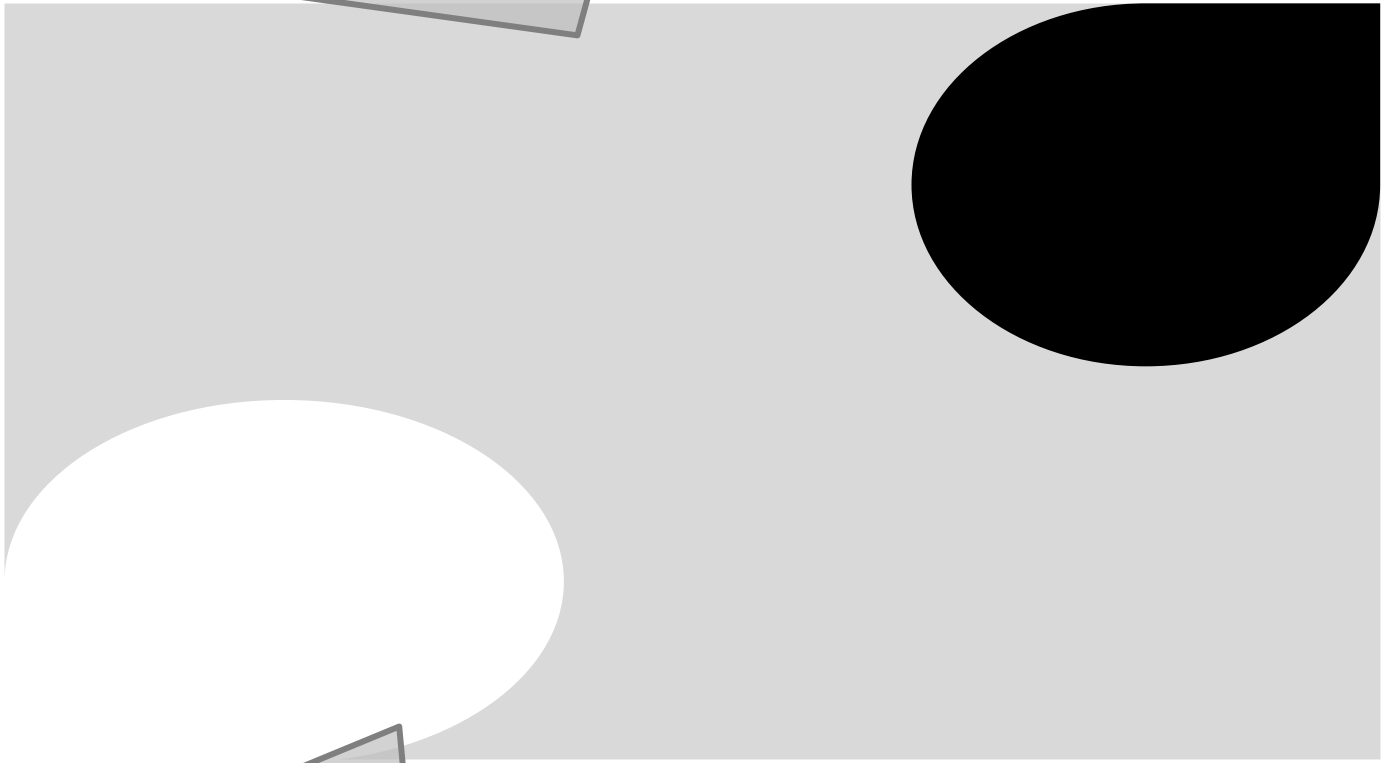
As testing is based on requirements, this leads to the following distinction:

- **positive** Requirements – What should the system do?
- **negative** Requirements – What should the system **not** do?

Note that these two terms are uncommon even though they would make sense. Negative requirements are often called „security requirements“, though not all negative requirements are about security (they may, for instance, also be about employee rights protection) and positive requirements are called … well, just „requirements“.

Positive and Negative Requirements

This square contains all functionality our software could possibly have. Some of this functionality is welcome, some is tolerable, some is definitely unwelcome.

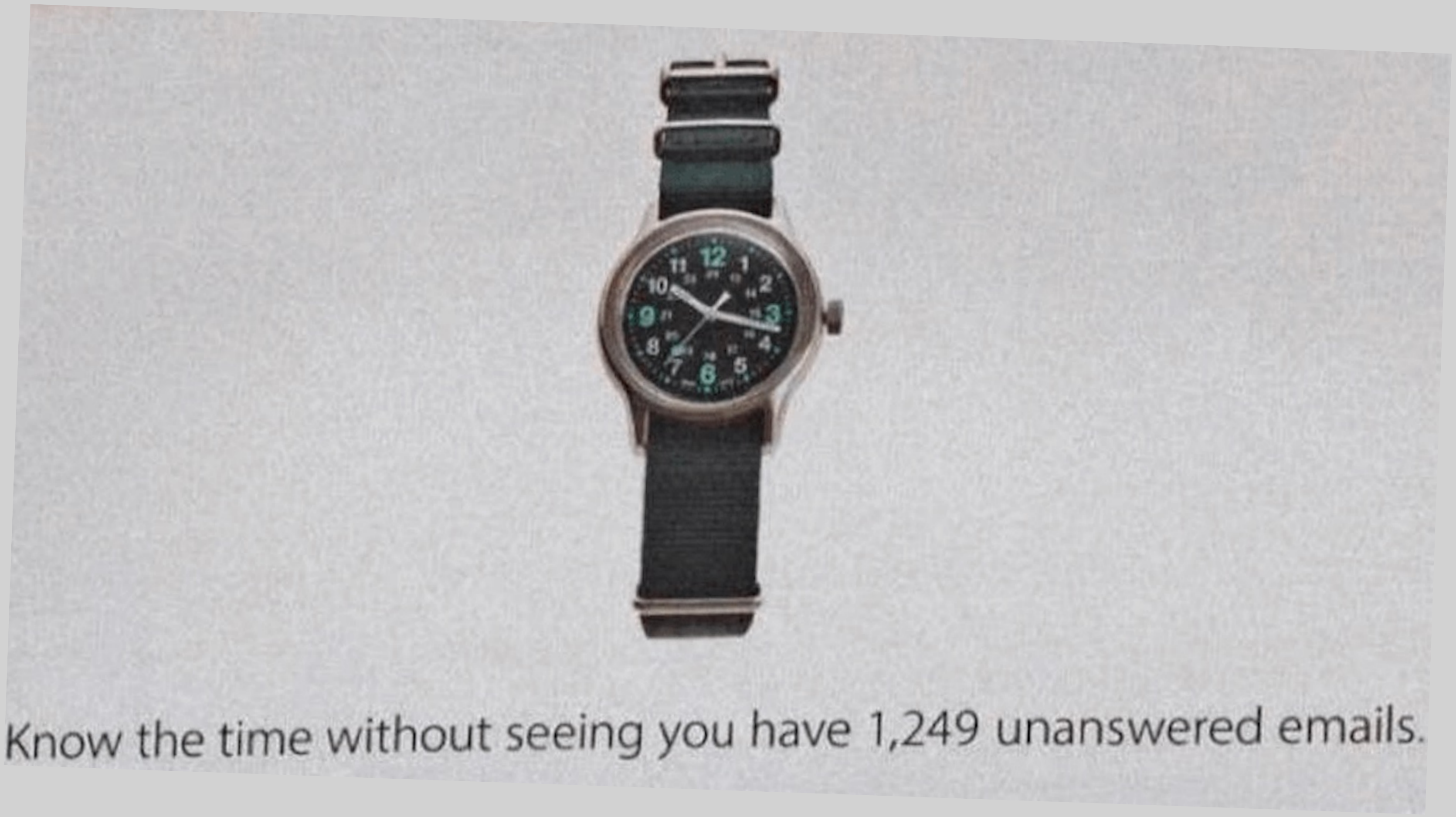


Negative requirements are about what the software must **not** do. But still, they are requirements. **To achieve quality, the negative requirements, too, must be known.**

Grey area: Behavior of the software that is OK, but not required and not prohibited.

Positive requirements are about what the software must do.

Negative Requirements: Example


Know the time without seeing you have 1,249 unanswered emails.

Another Way to Categorize Requirements - Exercise

Name a positive requirement for a burglar alarm system. Name a negative one.

- **Positive** Requirements – What should the system do?
  Sounding an alarm if a sensor goes off.

- **Negative** Requirements – What should the system **not** do?
  Sounding an alarm if a cat walks by („Kleintierunterdrückung" – probably one of the worst German words)

Content

Structuring...

The Whole Body of Requirements

Requirements Documents

Functional Requirements

Non-Functional Requirements

Requirements Backlogs

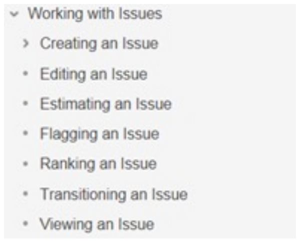
Individual Requirements

How are Functional Requirements Typically Structured?

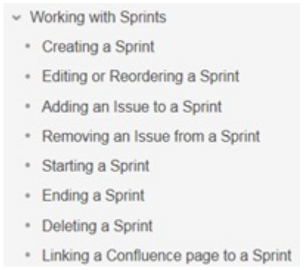
What are the usual sub-blocks of the block of the Functional Requirements? **Functional** Requirements (but also other software descriptions like user manuals) are often structured by **objects**.

By „object" we don't mean each technical object that we find in the source code. Instead, we mean objects that make up the application domain and that are visible to the user (often called **domain objects** or **business objects**).
In the example on the right hand side, the objects are *Issue* and *Sprint*.

User documentation is often structured by use case.



- Working with Issues
  - Creating an Issue
  - Editing an Issue
  - Estimating an Issue
  - Flagging an Issue
  - Ranking an Issue
  - Transitioning an Issue
  - Viewing an Issue



- Working with Sprints
  - Creating a Sprint
  - Editing or Reordering a Sprint
  - Adding an Issue to a Sprint
  - Removing an Issue from a Sprint
  - Starting a Sprint
  - Ending a Sprint
  - Deleting a Sprint
  - Linking a Confluence page to a Sprint

Each object is structured by **actions** (*creating*, *editing*, ...)
An action that leaves the system in a consistent state is called a use case. In the example on the right hand side, you see a list of **use cases ordered by domain object**.

This is a very frequent way to structure functional requirements.

List

Exercise: «EarlyBird»

«EarlyBird»
Your Online Breakfast Service

What are the domain objects of «EarlyBird»?

What are the use cases for each domain object?

Recap: Structuring Functional Requirements

A good way to structure functional requirements is

1. by **domain object** and
2. for each **domain** object by **use case**.

Customer:

- Create Customer
- Update Customer
- Search Customer
- Blacklist Customer
- ...

Order:

- Create Order
- Track Order
- Cancel Order
- Renew Order
- ...

More Precise Definition: What is a Use Case?

The UML standard defines a use case as a "**unit of useful functionality**". More specifically, a use case is an interaction with the system that is performed …

- by **one person or system** (therefore „Create BusinessPartner" and „Create Contract" are two use-cases. Those two steps are usually not performed by the same person)

- at **one time** (therefore „Create Contract" and „Release Contract" are two use cases - they need not necessarily happen at one time. The Contract can be released days after it was created)

More Precise Definition: What is a Use Case? / 2

- **adds business value** (therefore a logon is not a use case - at least not a primary one) and
- leaves the **data in a consistent state**.” (therefore the implementation of a use case typically ends with writing data persistently into the database - unless it is a read-only use-case)

Source: Craig Larman, Applying UML and Patterns, 2nd ed., p. 60

Naming Use Cases

Use cases typically are named
<Action> <DomainObject>,
e.g. *Create Customer* or *Cancel Order*.

An **analysis of thousands of real-world use cases** has found that the **actions shown on the right hand side are very frequent**.

Note that many objects have a status field with predefined values. For instance, a product could be *announced*, *available*, *reordered* or *discontinued*. This is often reflected directly in the use cases. So, in the example, there could be use cases like *Reorder Product* or *Discontinue Product*.

- Assign
- Calculate
- Check
- Collect
- Complete
- Convert
- Create
- Delete
- Freeze
- Generate
- Initialize
- IsCorrect
- IsExistent
- Maintain
- Migrate
- Read
- Receive
- Rectify
- Reject
- Renew
- Report
- Resolve
- Round
- Search
- Select
- Send
- Set
- SetStatus
- Start
- Stop
- Touch
- Transform
- Trigger
- Update

Scenarios

Let‵s look at the use case *Buy Product* of an online shop. The user may want

A. to redeem a voucher,
B. or to redeem no voucher,
C. to have the product gift-wrapped,
D. or not have it gift-wrapped.

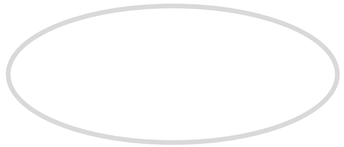
Note that **combinations are possible**, e.g. AC or BC, but not e.g. AB. Note also that you will probably **first implement BD** and later add A and C as further options.

AC, BC etc. are called „**scenarios**", i.e. certain **sequences of paths through the use case**. BD is a special scenario, because it is the „easiest path" through the use case. Sometimes, this scenario is called the **happy path**.

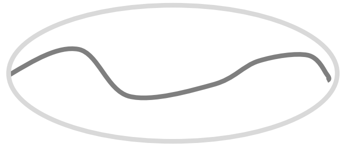
Two Types of Functional Requirements

The next four slides show how a system may evolve over time, iteration by iteration.

Use cases are shown as ellipses:

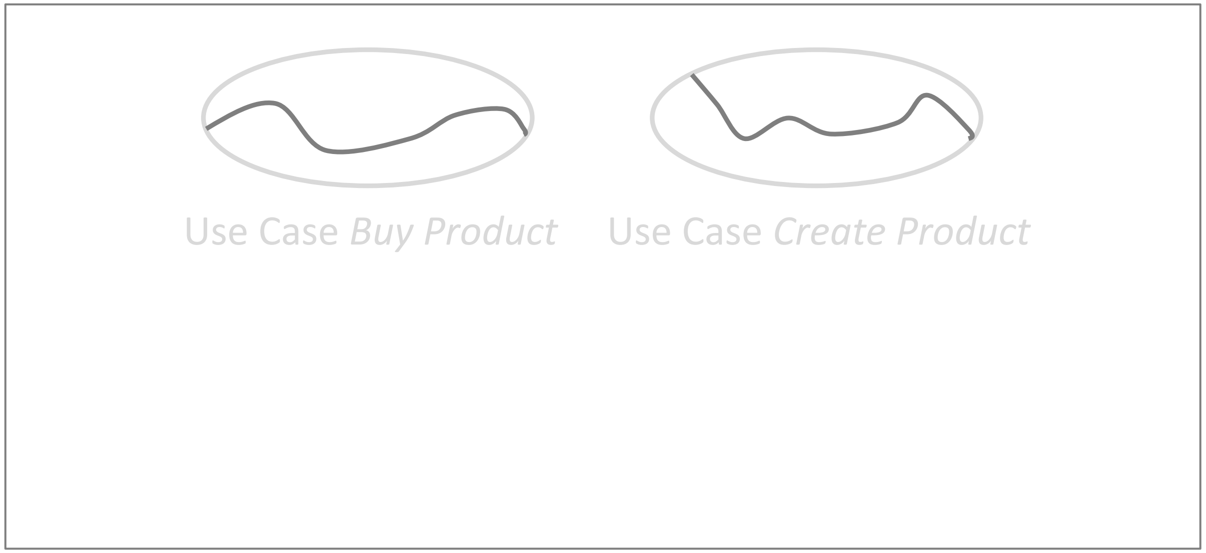


Scenarios of a use case are shown as wiggly lines:



The happy path of a use case is shown by a solid line, all other paths by dashed lines.

How the Software Evolves: Iteration 1

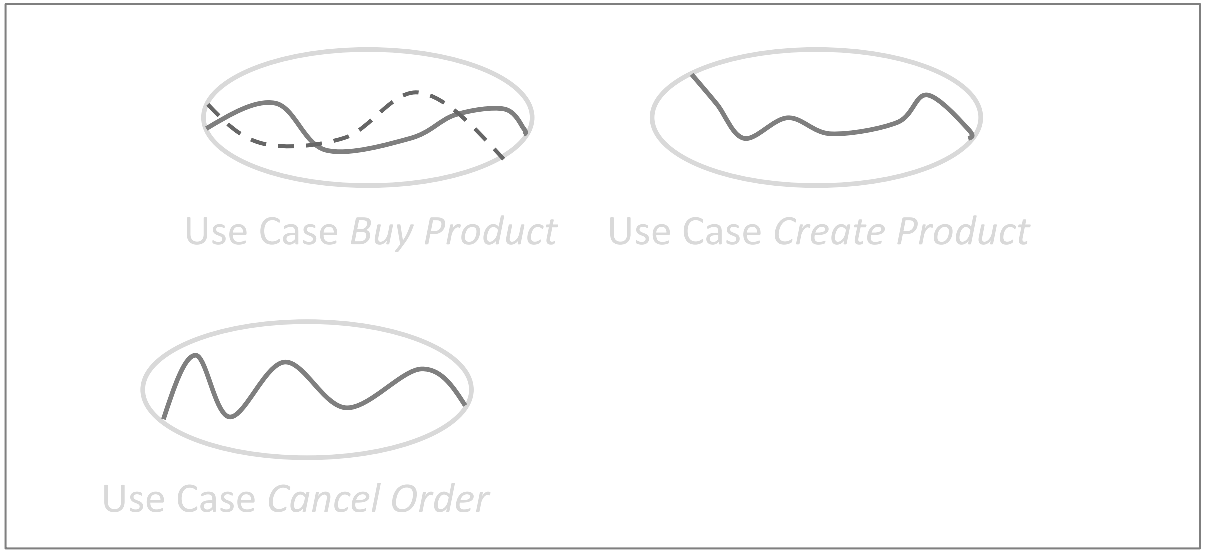


———	Happy Path
- - - - - -	Alternate Scenario
··············	Error Scenario

As a dealer, I want to create products so customers can buy them.

As a customer, I want to buy products so I can be happier.

How the Software Evolves: Iteration 2

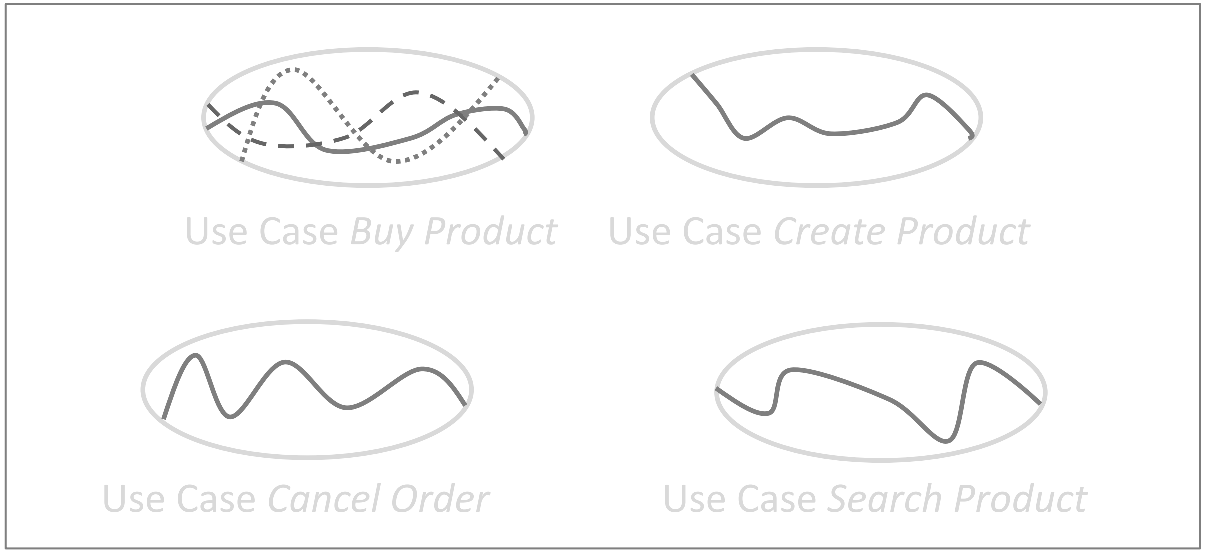


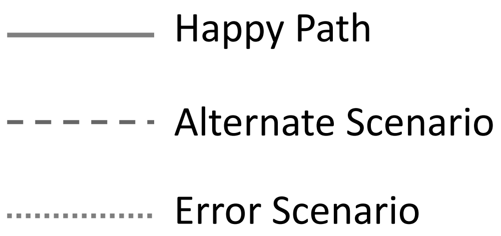
───	Happy Path
- - - -	Alternate Scenario
········	Error Scenario

As a customer, I want to cancel orders
so I can undo errors.

As a customer, *when buying a product*,
I want to gift-wrap the product
so I can use it as a present.

How the Software Evolves: Iteration 3

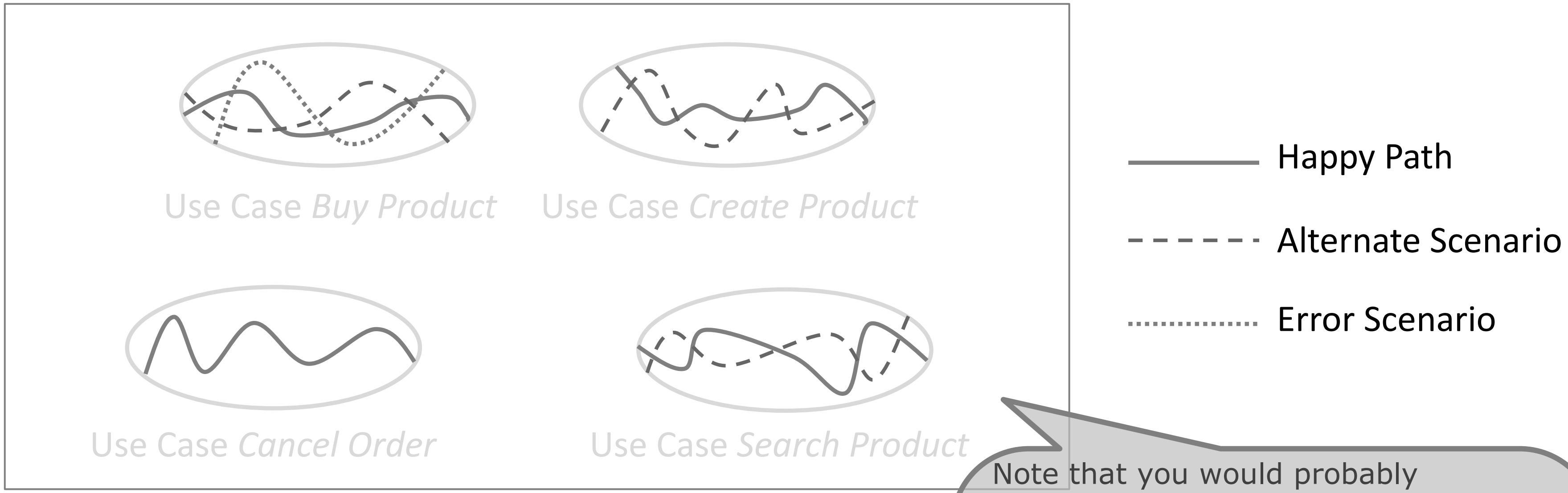




As a customer, I want to search products, so I don‘t need to remember product numbers.

As a customer, *when buying a product*,
I want to see an error message if my order value is greater than €10000
so I avoid going bankrupt.

How the Software Evolves: Iteration 4



As a dealer, *when creating a product*,
I want to add a manual as a pdf.

As a customer, *when searching for a product*,
I want be able to sort the result by price so I can find find the cheapest offer.

Note that you would probably structure the product documentation **by use case, and within each use case by scenario**:

1. Creating Products
    1.1 Adding attachments
2. Searching Products
    2.1 Result sorting
3. Buying Products
    3.1 Gift-wrapping
    3.2 Bankruptcy prevention
4. Canceling Orders

Essential and Non-Essential Use Cases

Sometimes the system offers the same use case across several technologies. For instance, the use case *List customers* could list the customers

- on the screen
- as a spreadsheet
- as a csv-file and
- as a pdf-file.

How many use cases are there? This is where the distinction between essential and non-essential use cases comes in. **Essential use cases are technology-free, non-essential ones are not.**

So if you do essential use case modeling, there is one use case here.

If you do non-essential use-case modeling, there are four.

Primary and Secondary Use Cases

Sometimes you want to describe a **part** of a use case, e.g. because several use cases have some part in common which you want to describe only in one place (which is an example of **requirements reuse** to avoid redundancy and adhere to DRY = Don’t Repeat Yourself).

Some examples:

- a check for creditworthiness,
- a login or
- entering a customer number.

A part of a use case that you describe separately is called **secondary use case.**

For distinction, a complete use case is sometimes called **primary** use case.

Two Types of Secondary Use Cases

Requirements reuse is not the only reason to use secondary use cases. The other reason is **exceptional** behavior, as opposed to **regular** behavior.

For operating a cell phone, regular behavior includes sending text messages. Exceptional behavior includes what should happen if the phone has no reception.

For depositing funds, regular behavior includes crediting the account. Exceptional behavior includes what should happen if the money laundering check comes back with a positive result.

It has proven advantageous to **keep the requirements for exceptional behavior separate from the requirements for regular behavior**. One reason is that very likely they will not be implemented at the same time. In an agile environment, you will probably start by implementing the regular behavior.

Use Cases and Software Design

One advantage of identifying use cases is that it **helps software design**. There are a number of design approaches (Clean Architecture, Hexagonal Architecture, …) that propose that you **create a distinct class for each use case.**

Ian Cooper, for instance, says: “*Controllers are objects that implement use cases. Name them after the use case”,* see „DevTernity 2019: Ian Cooper – The Clean Architecture“, https://www.youtube.com/watch?v=SxJPQ5qXisw, at 30:58.

On the same issue, Robert C. Martin says that the essence of the work of Ivar Jacobson on software design is *„You can take a use case and turn it into an object.*“ Jacobson calls those objects *Interactors*.

See „The Principles of Clean Architecture by Uncle Bob Martin“, https://www.youtube.com/watch?v=o_TH-Y78tt4, at 20:14.

Use Cases and Testing

Identifying use cases also helps testing: “*There is a correlation between the use case boundary and the test boundary – tests should focus on the behavior expressed by a use case*.“ (see „DevTernity 2019: Ian Cooper – The Clean Architecture“, https://www.youtube.com/watch?v=SxJPQ5qXisw, at 33:10).

How to Describe a Use Case

So functional requirements are often described in the form of use cases. But what is the best way to describe a use case?

Typically, a **use case template** is used. **But what makes a good use case template?**

We will explore this question in the next slides.

Use Case Specification Templates

Based on the file „Use Case Specification Template‟ (see separate file), write a use case specification for the Use Case of withdrawing money from an ATM.

Add a diagram
that improves clarity the most.
It need not be UML.

What Use Cases Don‘t Tell Us

- Use cases are a way to structure **functional** requirements. They tell us very little about non-functional ones.
- Use cases are a way to describe what the system can do, but **not what it can‘t do**.
- Knowing what use cases a system offers is a far cry from knowing how to use it. To use a system properly, often several users have to perform several use cases in a predefined **order**. In other words: A **business process** typically consists of a number of **use cases in a given order**.

Example of a business process:

1. In the morning, the clerk performs Load ATM.
2. During the day, customers perform Withdraw Money.
3. In the evening, the clerk performs Print Withdrawal List.

Content

Structuring...

Types of Non-Functional Requirements

There are many types of non-functional requirements, which is why some of them are easily missed in a software project.

This is why Christian Kästner recommends:

„Requirements taxonomies and checklists can be used across projects to ensure that common requirements are at least considered."

Note 1: A **taxonomy** is a classification scheme.

Note 2: Non-Functional Requirements are often called **quality requirements**.

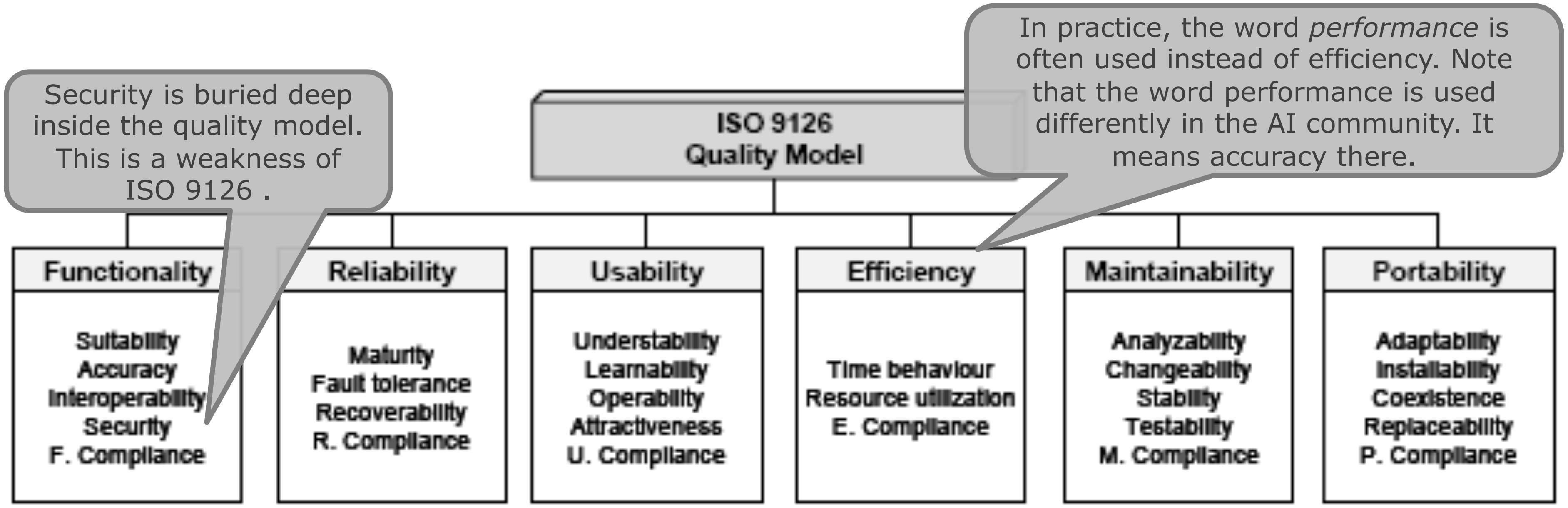
Let‘s look at some of those taxonomies.

Source: Christian Kästner, Machine Learning in Production: From Models to Products, The MIT Press, 2025

Types of Non-Functional-Requirements - ISO 9126

„Requirements taxonomies and checklists can be used across projects to ensure that common requirements are at least considered.“

Christian Kästner, Machine Learning in Production: From Models to Products, The MIT Press, 2025

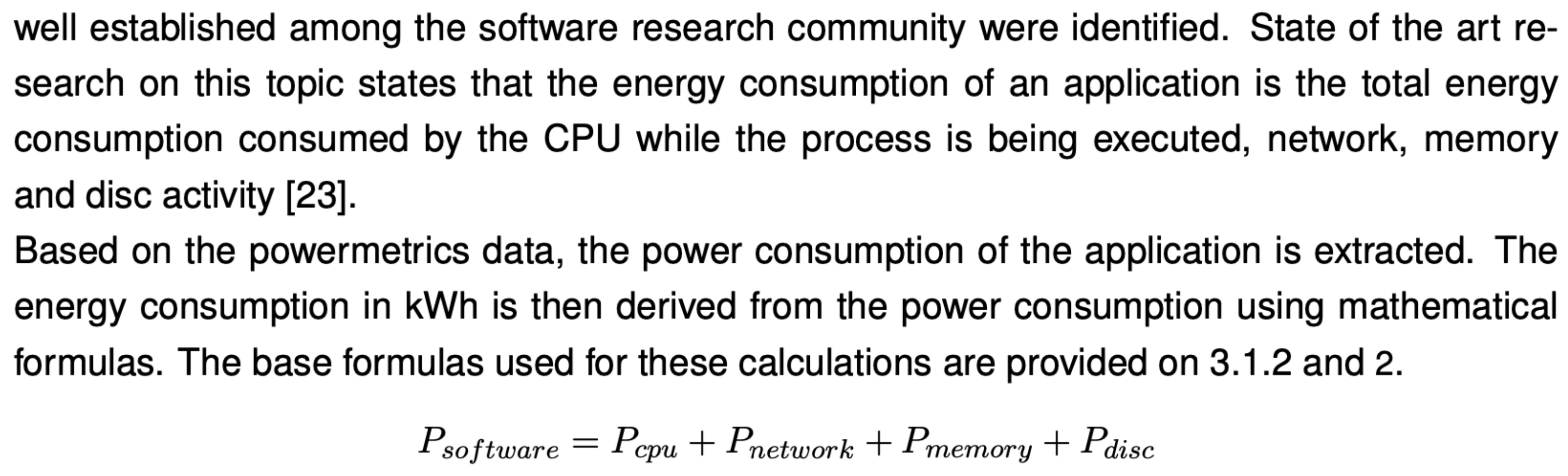


Note that ISO 9126 has been withdrawn and replaced by ISO 25000.
Still, ISO 9126 is in widespread use.

Source: http://www.jot.fm/issues/issue_2007_10/paper22/

Energy Efficiency

Today, **being energy-efficient is a major non-functional requirement placed on all kinds of software**. Here is some research on how to build software that consumes less energy.



well established among the software research community were identified. State of the art research on this topic states that the energy consumption of an application is the total energy consumption consumed by the CPU while the process is being executed, network, memory and disc activity [23].
Based on the powermetrics data, the power consumption of the application is extracted. The energy consumption in kWh is then derived from the power consumption using mathematical formulas. The base formulas used for these calculations are provided on 3.1.2 and 2.

$$P_{software} = P_{cpu} + P_{network} + P_{memory} + P_{disc}$$

Source: Bjorna Kalaja, Applying refactoring techniques to improve environmental impact of a software application, 2024

Content

Structuring...

List

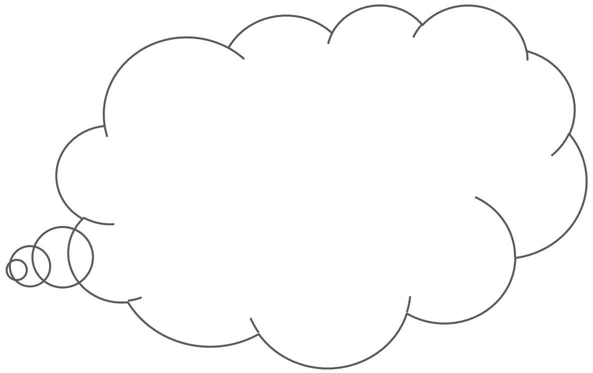
Structuring the «EarlyBird» -Requirements

«EarlyBird» is licenced by another customer. The contract should contain a document that lists the requirements fulfilled by the software. The current requirements list is not suitably structured.

«EarlyBird»
Your Online Breakfast Service

Propose a suitable table of contents for the «EarlyBird» requirements. You don’t need to write the whole document, just the table of contents.

First, try it on your own.
Then using AI.
Then with an improved prompt.

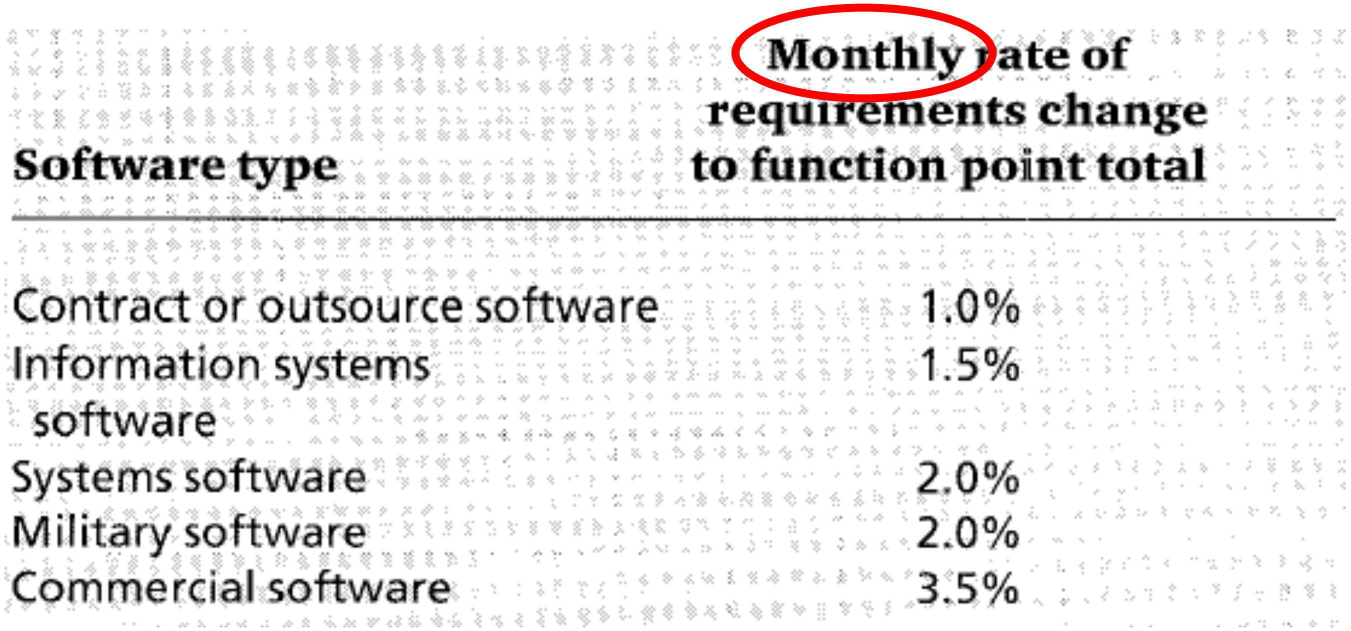


Content

Structuring...

Requirements Change is a Fact of Life

Many weaknesses of requirements documents have to do with the fact that they are **hard to keep up-to-date**.
That is a serious problem because software requirements change **really fast**.



Software type	Monthly rate of requirements change to function point total
Contract or outsource software	1.0%
Information systems software	1.5%
Systems software	2.0%
Military software	2.0%
Commercial software	3.5%

Source: Capers Jones, Strategies for Managing Requirements Creep, http://www.students.science.uu.nl/~3092062/papers/11.PDF

The Top-10 RE Problems in Practice

In a perfect world, we have **one** customer who provides us with high-quality requirements. In practice, however, we need to cope with questions like ...

1. How can we find **all** sources of requirements?
2. What can we do if the requirements sources disagree?
3. How can we elicit the requirements the customer has but doesn't tell us?
4. How can we find out if the customer **really** needs what she says?
5. What is the best prioritization method for requirements?
6. What is the best table of contents for a big requirements document?
7. What can we do to get **precise** natural language requirements?
8. What are the best diagram types for visualizing requirements?
9. What are the most important requirements attributes and relationships?
10. What can we do to keep the requirements up-to-date for years?

Content

Structuring…

The Whole Body of Requirements

Requirements Documents

Requirements Backlogs

Individual Requirements

Documents and Backlogs

A requirement that is part of a backlog is called **backlog item**.

"Contrary to popular misunderstanding, the Product Backlog does ***not*** *contain "Stories";* ***it simply contains items****. Those items can be expressed as Stories, use cases, or any other requirements approach that the group finds useful*."

Note that a **large** backlog item is often called an **Epic**.

Source (quote): Pete Deemer; Gabrielle Benefield; Craig Larman; Bas Vodde . "The Scrum Primer: A Lightweight Guide to the Theory and Practice of Scrum (Version 2.0)", 2018

Backlog Items – Quality Criteria

Typically, before the start of each iteration (or in Scrum terminology: sprint), you take the top items of the backlog and plan to implement them in the iteration.

That approach will not work, however, if the backlog items don‘t fulfil certain **quality criteria**. For instance, they need to be

- **Independent**, because otherwise you can‘t just take the top items, since they possible need the implementation of lower-ranked backlog items first.
- **Small**, because otherwise they won‘t fit into the iteration.

Those are just two examples of quality criteria for backlog items. There is a well known set of them: the **INVEST-criteria**. **They apply to all kinds of requirements**, whether they are in a backlog or not.

Backlog Items – Quality Criteria / 2

A backlog item is well written if it satisfies the **INVEST**-criteria:

Independent: You could implement the backlog items in any arbitrary order. Or you could implement only some of them. Regardless of what you do, the software always works.

This brings us back to the rules of dependency management, the first of which was: **Avoid dependencies!**

Backlog Items – Quality Criteria / 3

Negotiable: The description of the backlog item does not cover all the details. The details are worked out with the customer during implementation.

Valuable: The backlog item is valuable to the customer. "Re-index the table RG308X" is no good backlog item, because its value to the customer is unclear.

V may also be interpreted as „**vertical**": The backlog item covers database, business logic **and** GUI at the same time, because only then the customer gets the value (= is able to use the backlog item).

„Add field *Delivery Preference* to Customer Screen" is not a good backlog item, because if the field is not stored in the database and processed by the business logic, it is not valuable to the customer.

Backlog Items – Quality Criteria / 4

E**stimable**: Note that this needs a certain depth of description.

S**mall**: The item fits into a small development cycle, sometimes as short as one week.

T**estable**: Again, this needs a certain depth of description.

Summary

So a backlog is an **ordered** list of requirements that satisfy the **INVEST-criteria**:

- **I**ndependent
- **N**egotiable
- **V**aluable/Vertical
- **E**stimable
- **S**mall
- **T**estable

But how can we order them? This topic will be explored on the following slides.

The Top-10 RE Problems in Practice

In a perfect world, we have **one** customer who provides us with high-quality requirements. In practice, however, we need to cope with questions like ...

1. How can we find **all** sources of requirements?
2. What can we do if the requirements sources disagree?
3. How can we elicit the requirements the customer has but doesn't tell us?
4. How can we find out if the customer **really** needs what she says?
5. What is the best prioritization method for requirements?
6. What is the best table of contents for a big requirements document?
7. What can we do to get **precise** natural language requirements?
8. What are the best diagram types for visualizing requirements?
9. What are the most important requirements attributes and relationships?
10. What can we do to keep the requirements up-to-date for years?

Requirements Priorities in Practice

In practice, requirements prioritization does **not** work
if two requirements may have the **same** priority.
The customer will award the highest priority level to many,
if not most requirements.

We have to make the customer choose:
This is why there are usually **no ties** in backlogs.

Content

Structuring...

The Whole Body of Requirements

Individual Requirements

How Structured are Natural Language Requirements?

What goes for LLM outputs also goes for requirements. Of course, requirements documents written in natural languages like English or German are **not completely free-form** in the first place.

Rather, natural languages have

- pre-defined **word types** (namely nouns, verbs, adjectives, ...),
- pre-defined rules how to **connect** words, e.g, that in English the adjective is put before the noun, not after it and
- pre-defined **attributes** for words and sentences.

A document that describes word types, connections and attributes (among other things) of a language is usually called a **grammar.**

Are Unstructured Textual Requirements Good?

For requirements reviews, requirements structuring, test case generation and many other requirements-based activities, using a **natural** language is often **not** considered to be **strict enough**.

Therefore, requirements engineers often introduce an **artificial** language with a **stricter grammar**, i.e. stricter rules concerning word types, connections as well as word and sentence attributes. Note that a grammar of an artificial language is often called a **metamodel**.

This is why Harry Sneed says:

Unstructured text is one of the **worst forms** that requirements can have.
*„Requirements **must** be underlaid with a **metamodel**.“*

In other words: **They must have tags.**

Source (quote): Harry Sneed, personal communication, 2019

Content

Structuring...

The Whole Body of Requirements

Individual Requirements

More Word Types

Stricter Connection Rules

Attributes

Special Types of Words in Requirements Engineering

What special word types would be so interesting for requirements engineering that they should get tagged?

For instance,

- words that name **actions** that the system under development shall perform,
- words that name **interfaces** that the system shall have or
- words that name **messages** that the system shall send or receive,

See the next slide for more examples.

Special Word Types for Requirements Engineering A Real-World Example

Tag	Definition	Examples
domain object	Object (more precisely: class) from the application domain	Order, Customer, Product, ...
attribute	Property of a domain object	Order.number, Customer.age
use case	Self-contained interaction with the system.	Create Customer, Search Product
state	State a domain object can be in	Open, closed, locked
condition	Requirement containing a condition	Children pay half the price

Beacons

If we have a typical unstructured requirements text, how can we add structure to it using the special word types?

One approach is to introduce **beacons** (German: *Leuchtfeuer*) into the requirements text:

You read through the requirements text and label the words that fit into one of the special categories. So, for instance,

- if you come across a word like *product*, you label it as a domain object,
- if you come across a word like *cancellation*, you label it as a use case,
- ...

Beacons Example: Health File

This is what the raw requirements text looks like:

"A health file may only be passed on to non-medical personnel if the patient has agreed in writing and the patient's age is over 17. The patient also has the option to set their health file's confidentiality status to OPEN, in which case the above-mentioned written agreement is not necessary."

This is what the text looks like once beacons have been added:

"A <use case> <attribute> health </attribute> <domain object> file </domain object> may only be passed on </use case> to <domain object> <attribute> non-medical </attribute> personnel </domain object> <condition> if the <domain object> patient </domain object> has <use case> agreed <attribute> in writing </attribute> <use case> and the patient's <attribute> age </attribute> is over 17 </condition>. The <domain object> patient </domain object> also has the option to <use case> set their <attribute> health </attribute> <domain object> file's </domain object> <attribute> confidentiality status </attribute> to <state> OPEN </state> </use case>, <condition> in which case the above-mentioned <attribute> written</attribute> <domain object> agreement </domain object> is not necessary </condition>."

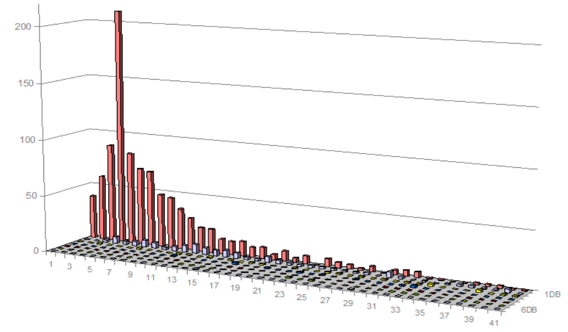
What are Beacons Good for? They Provide a Starting Point for …

Structuring the body of functional requirements into domain objects and use cases as we did in the previous chapters.

System modeling: Suppose you want to design a

- use case diagram: then you look for <use case>
- class diagram: then you look for <domain object>
- state machine for health files: then you look for <state>

Effort estimation: **The beacons are the effort drivers**. In a multi-variate analysis across hundreds of requirements, NumberOfUseCases + 3,5*NumberOfClasses turned out to be best correlated with the actual effort in person-days.



Exercise: «EarlyBird»

«EarlyBird»
Your Online Breakfast Service

Add beacons to the first 20 lines of the requirements text of «EarlyBird» similar to the approach shown in the health file example.

You should be able to **present** your results.

Exercise: «EarlyBird»

«EarlyBird»
Your Online Breakfast Service

If you were to add another three types of beacons, what would those be?

Tooling Requirements

As a effort estimator (among other roles), I want a tool that takes a group of requirements as its input and adds beacons so that I can estimate the effort more easily (among other benefits).

Content

Structuring...

User Stories

Natural language has its rules for connecting words.
For instance, *„I chocolate eating like.“* is no proper English sentence.

It is a frequent practice in requirements engineering to use **stricter** rules for ordering words in requirements. Those rules are often enforced by **templates**.

One of the most popular templates is the user story template:
As a < type of user >, I want < some goal > so that < some reason >.

- *As a bank customer I want to withdraw money from an ATM so that I can get cash whenever needed*.
- *As bank customer I want to choose the denomination (German: Stückelung) of the bills so I can use the bills more easily.*

User Stories and Non-Functional Requirements

Note that user stories are not only suitable for functional requirements, but **also for non-functional requirements**:

As a broker, I want to have all functionality on my cell phone so that I can stay informed everywhere.

As a person with impaired vision, I want to have the screen display read out loud to me so that I don‘t need to strain my eyes.

Use Cases and User Stories

A frequent question is: **What is the relationship between use cases and user stories?** A typical user story either

- introduces a **whole new use case** to the system (typically by only implementing its happy path, not sophisticated scenarios)
  Example: *As a bank customer I want to withdraw money from an ATM so that I can get cash whenever needed*.
  This user story adds the new use case *Withdraw Cash*. Before that, the ATM simply did not have that use case.

- or adds **new scenarios** to an existing use case.
  Example: *As bank customer, when withdrawing cash, I want to choose the denomination of the bills so I can use the bills more easily.*
  This user story adds a new scenario to the existing use case *Withdraw Cash*.

Use Cases and User Stories / 2

This connection between use cases and user stories is why Martin Fowler writes ***"A story might be one or more scenarios in a use case […]"***

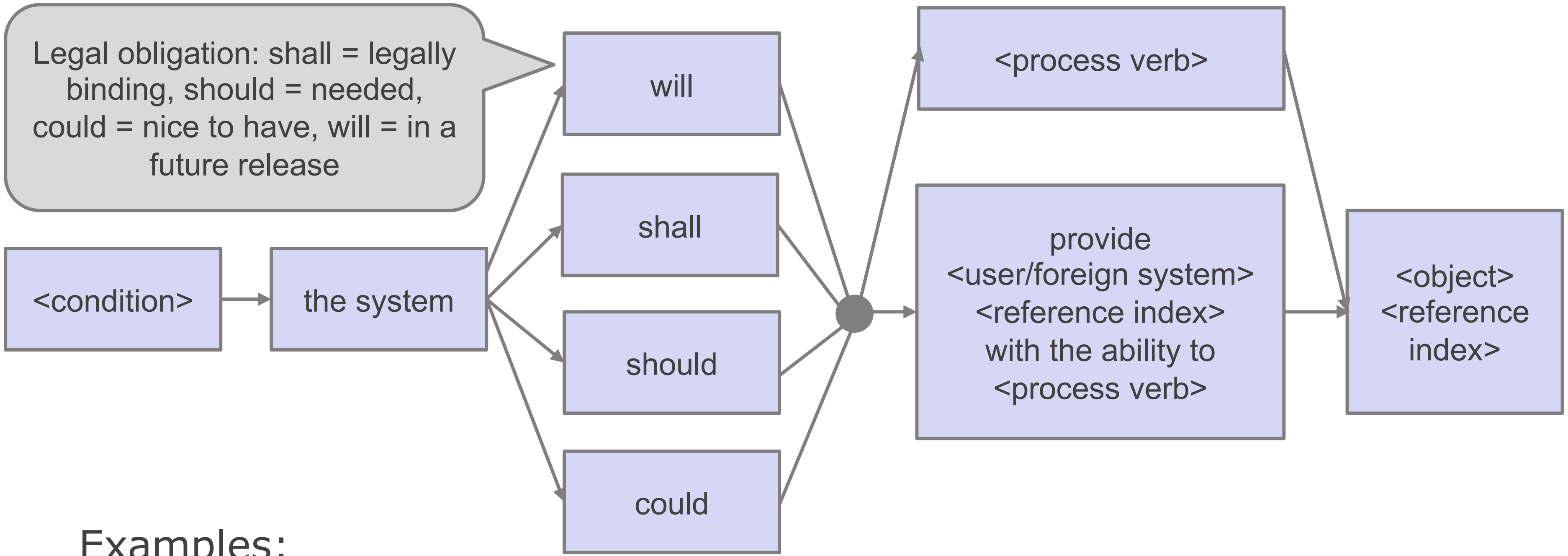
Note that

- there are **dependencies** between those two types of stories: You can't implement choosing denominations before implementing withdrawing cash. There is no point in adding scenarios to a non-existing use case.
- for stories that add scenarios, the template As a < type of user >, **when <use case>**, I want < some goal > so that < some reason > is more precise: *As bank customer, **when withdrawing money**, I want to choose the denomination of the bills so I can use the bills more easily.*

Source (quote): M. Fowler, https://martinfowler.com/bliki/UseCasesAndStories.html, 2017.

Another Requirements Template

The user story template is not the only requirements template.
Here is an alternative that was used in a public transport project:


Legal obligation: shall = legally binding, should = needed, could = nice to have, will = in a future release
<condition>
the system
will
shall
should
could
<process verb>
provide <user/foreign system> <reference index> with the ability to <process verb>
<object> <reference index>

Examples:

- Two weeks before the expiry date, the system should send notification emails to holders of railway classic gold cards.
- If a customer pays for a ticket, the ticketing system shall be able to provide the payment system with the payment data.

Exercise: «EarlyBird»

«EarlyBird»
Your Online Breakfast Service

Propose an **optimal requirements template** for «EarlyBird» that is **not** simply one of the ones described above.

Re-write at least seven requirements of «EarlyBird» to fit into your template.

You should be able to **present** your results.

Content

Structuring...

Requirements Attributes

A related question concerns **requirements attributes**. We have already encountered several of those attributes, e.g.

- Customer value
- Cost
- Risk
- Rank in the backlog

On top of those, are there other meaningful requirements attributes?

Note that relationships between requirements bear a close resemblance to requirements attributes, because, effectively, they are special attributes containing references to other requirements.

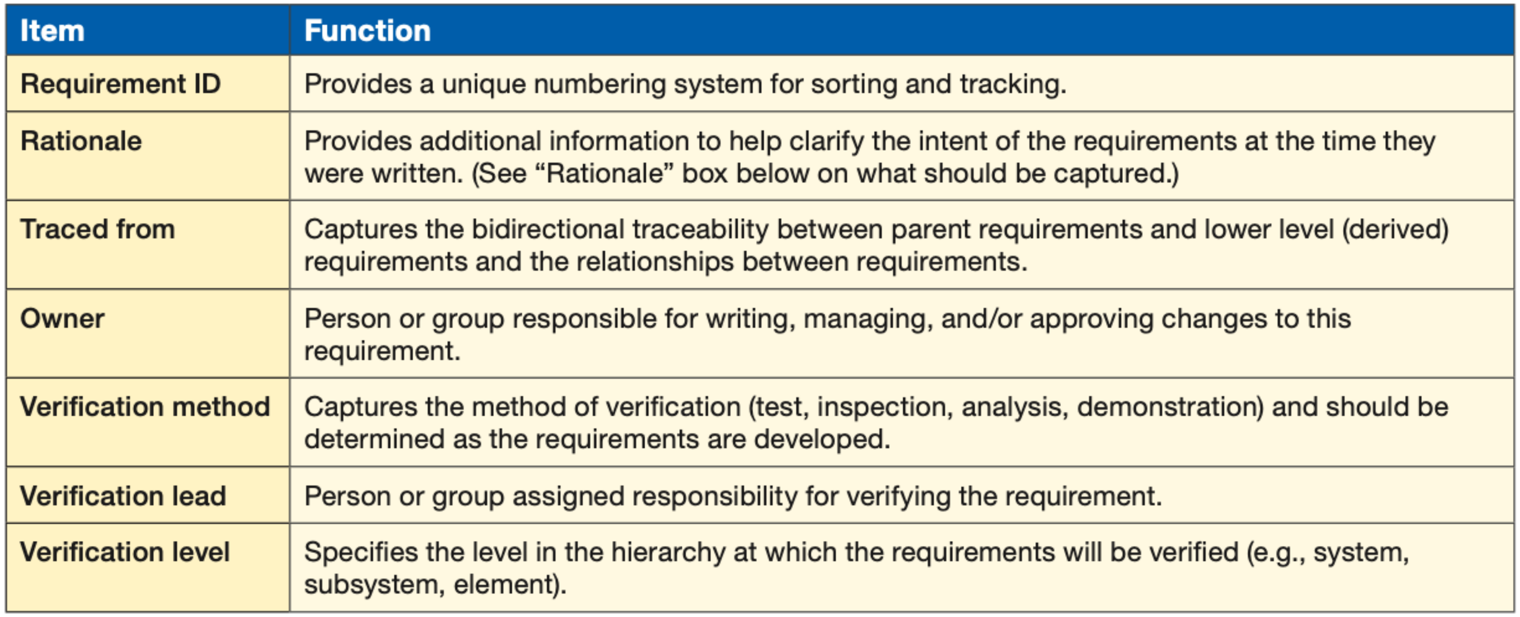
The Top-10 RE Problems in Practice

In a perfect world, we have **one** customer who provides us with high-quality requirements. In practice, however, we need to cope with questions like ...

1. How can we find **all** sources of requirements?
2. What can we do if the requirements sources disagree?
3. How can we elicit the requirements the customer has but doesn't tell us?
4. How can we find out if the customer **really** needs what she says?
5. What is the best prioritization method for requirements?
6. What is the best table of contents for a big requirements document?
7. What can we do to get **precise** natural language requirements?
8. What are the best diagram types for visualizing requirements?
9. What are the most important requirements attributes and relationships?
10. What can we do to keep the requirements up-to-date for years?

Requirements Attributes

TABLE 4.2-2 Requirements Metadata



Item	Function
Requirement ID	Provides a unique numbering system for sorting and tracking.
Rationale	Provides additional information to help clarify the intent of the requirements at the time they were written. (See "Rationale" box below on what should be captured.)
Traced from	Captures the bidirectional traceability between parent requirements and lower level (derived) requirements and the relationships between requirements.
Owner	Person or group responsible for writing, managing, and/or approving changes to this requirement.
Verification method	Captures the method of verification (test, inspection, analysis, demonstration) and should be determined as the requirements are developed.
Verification lead	Person or group assigned responsibility for verifying the requirement.
Verification level	Specifies the level in the hierarchy at which the requirements will be verified (e.g., system, subsystem, element).

Source: https://www.nasa.gov/wp-content/uploads/2018/09/nasa_systems_engineering_handbook_0.pdf, 2025

Assigning Attributes to Requirements

Requirements have attributes (in the most simple case a description and an identifier like a requirements number). Attributes help to manage requirements over their whole lifecycle.

But what attributes do we need to manage requirements over their life cycle? Core attributes include

- **Identifier** (must be **unique**)
- **Description** (often a short one and a long one)
- **Status** (specified, rejected, …)
- **Source** (e.g. a stakeholder)
- **Stability** (how likely is it that the requirement will change in the future? This is needed for software design.)
- **Criticality** (how big is the damage to business and/or reputation if the requirement is not implemented?)
- **Priority** (should the requirement be implemented earlier or later?)

Deep Dive

On the following slides we take a deeper dive into some requirements attributes, namely

- **Identifier**

- **Conditions of Satisfaction**

- **Acceptance Criteria**

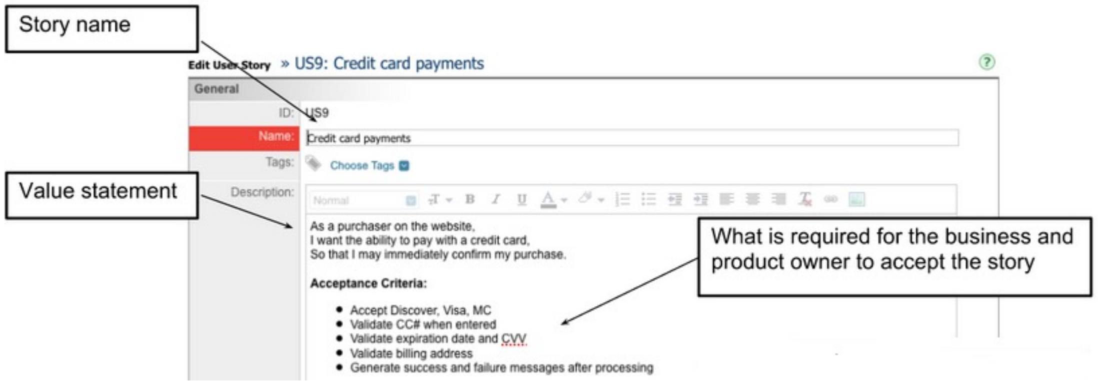
Requirement ID

Each requirement must have an ID.

Regardless of how you organize the requirements, at the lowest level it must consist of individual elements each of which has a unique, immutable (= does not change over the lifetime of the requirement) ID.

Otherwise you are not able to communicate properly about requirements („*Requirement 245 is done, but Requirement 242 is not done yet*.“).

In the example below, criteria don‵t.
So there is room
for improvement.


Story name
Value statement
What is required for the business and product owner to accept the story

Edit User Story » US9: Credit card payments

General

ID: US9

Name: Credit card payments

Tags: Choose Tags

Description:

As a purchaser on the website,
I want the ability to pay with a credit card,
So that I may immediately confirm my purchase.

Acceptance Criteria:

- Accept Discover, Visa, MC
- Validate CC# when entered
- Validate expiration date and CVV
- Validate billing address
- Generate success and failure messages after processing

Conditions of Satisfaction

A condition of satisfaction is *"a **high-level acceptance test** that will be true after the agile Story is complete."* (Mike Cohn)

So, if your user story is "*As a vice president of marketing, I want to select a holiday season to be used when reviewing the performance of past advertising campaigns so that I can identify profitable ones*."

... then your conditions of satisfaction could be:

- *Make sure it works with major retail holidays.*
- *Support holidays that span two calendar years (none span three).*
- *Holiday seasons can be set from one holiday to the next (such as Thanksgiving to Christmas).*
- *Holiday seasons can be set to be a number of days prior to the holiday.*

Acceptance Criteria

[An acceptance criterion is] “***list of expectations*** *for a specific Product Backlog Item as defined by the Product Owner prior to the beginning of a Sprint*” (https://www.agilealliance.org/definition-done-user-stories/, 2018)

Acceptance criteria are derived from the conditions of satisfaction by making them more concrete. For instance, you could take the condition of satisfaction *Make sure it works with major retail holidays* and derive the following acceptance criteria from it:

- *It works on Christmas.*
- *It works on Easter.*
- *It works on New Year’s Day.*

Gherkin

If you take things to the next level and refine Acceptance Criteria even further, you could use Gherkin, a business readable, yet formal language.

Main elements of Gherkin are
GIVEN precondition
WHEN action/trigger
THEN expected reaction of the system
BUT (optional): non-reaction of the system

Continuing the above example, we could write in Gherkin:
GIVEN Easter Sunday is on April 9th
WHEN user selects Easter
THEN user gets the performance numbers for April 8th and 9th
BUT user does not get numbers for April 7th (Good Friday)

Traceability

Some important requirements attributes have the purpose of supporting **traceability**. In the context of requirements engineering, traceability is the ability to trace a requirement

(1) back to its origins, i.e. the requirements sources. This is called **pre-RS traceability**. RS stands for requirements specification.

(2) forward to its implementation in design and code, also called **post-RS traceability**

(3) to requirements it depends on (and vice-versa). Sometimes you can only implement a requirement B if you have implemented another requirement A. (Example: A "There must be a dialog to enter the delay penalty for a delivery." B: "If a delivery has a delay penalty > 0, then …") This is called **traceability between requirements**.

Benefits of Requirements Traceability

The benefits of requirements traceability are:

- **Simpler verification**. If you want to test a certain requirement, you find the test cases easily via the traceability links.
- **Identification of unneeded requirements**. If a requirement can't be linked to any requirements source and any rationale, why have it in the project?
- **Identification of unneeded properties** in the system. If something (e.g. a software package) can't be linked to a requirement, why have it in the project?
- **Support of impact analysis**. Your customer changes a requirement. What packages, test cases and documentation chapters need to be changed along with it? If you have traceability, you can answer that question easily.

Implementation of Traceability

Traceability can be represented in various ways including

- **text-based references** (e.g. a comment “see requirement 4525” in the code)
- **hyperlinks**
- **trace matrices** (e.g. a spreadsheet. Column A: Requirements Number Column B: Test Case Number)
- **trace graphs** (e.g. a picture showing requirements and test cases as bubbles and links between them)

Example of Requirements Attributes: Volere

Volere is a well-known reference structure that describes requirements in so-called **snow cards**. Each snow cards contains

- Requirement number
- Type of the requirement
- Use cases the requirement refers to
- Description
- Requirements source
- Requirements rationale
- Two priorities as described in the Kano Model
- Traceability information
- Information how the requirement can be verified

Exercise: «EarlyBird»

«EarlyBird»
Your Online Breakfast Service

What requirements attributes and requirements relationships should «EarlyBird» have?