Certified Professional for Requirements Engineering

Foundation Level

Handbook

Martin Glinz
Hans van Loenhoud
Stefan Staal
Stan Bühne

Terms of Use

All contents of this document, especially texts, photographs, graphics, diagrams, tables, definitions and templates, are protected by copyright. Copyright © 2022 for this handbook is with the authors. All (co-)authors of this document have transferred the exclusive right of use to IREB e.V.

Any use of the handbook or its components, in particular copying, distribution (publication), translation, or reproduction, requires the prior consent of IREB e.V.

Any individual is entitled to use the contents of the handbook within the scope of the acts of use permitted by copyright law, in particular to quote these correctly in accordance with recognized academic rules.

Educational institutions are entitled to use the contents of the handbook for teaching purposes under correct reference to the work.

Use for advertising purposes is only permitted with the prior consent of IREB e.V.

Acknowledgements

The content of this handbook was reviewed by Rainer Grau, Karol Frühauf, and Camille Salinesi. Tracey Duffy performed an English review. Stan Bühne and Stefan Sturm did the final editing.

Version 1.0.0 was approved for release on November 11, 2020 by the IREB Council upon recommendation of Xavier Franch and Frank Houdek.

Version 1.1.0 was approved for release on August 15, 2022 by the IREB ExCo.

We thank everybody for their involvement.

Foreword

This handbook provides an introduction to Requirements Engineering based on the syllabus version 3.0 for the Certified Professional for Requirements Engineering (CPRE)—Foundation Level according to the IREB standard. It complements the syllabus and addresses three groups of readers:

- *Students and practitioners* who want to learn about Requirements Engineering and take the certification exam can use this handbook as a companion book to training courses offered by training providers, as well as for self-study and individual preparation for the certification exam. This handbook may also be used to refresh existing knowledge about Requirements Engineering, for example, when preparing for a CPRE Advanced Level course and exam.
- *Training providers* who offer trainings on the CPRE Foundation Level can use this handbook as a complement to the syllabus for developing their training materials or as a study text for the participants in their trainings.
- *Professionals* in industry who want to apply proven RE concepts and knowledge in their practical work will find a wealth of useful information in this handbook.

This handbook also provides a link between the syllabus, which lists and explains the learning objectives, and the literature on Requirements Engineering. Every chapter comes with references to the literature and hints for further reading. The structure of the handbook matches the structure of the syllabus.

The terminology used in this handbook is based on the CPRE Glossary of Requirements Engineering Terminology [Glin2020]. We recommend downloading this glossary from the IREB website and use it as a terminology reference.

You find more information about the CPRE certification program, including the syllabi, glossary, examination regulations and sample exam questions on the IREB website at https://www.ireb.org.

Both the authors and IREB have invested a significant amount of time and effort into preparing, reviewing and publishing this handbook. We hope that you will enjoy studying this handbook. If you detect any errors or have suggestions for improvement, please contact us at info@ireb.org.

We would like to thank all people who contributed to the creation and publication of this handbook. Karol Frühauf, Rainer Grau and Camille Salinesi carefully reviewed the manuscript and provided valuable suggestions for improvement. Tracey Duffy did an English review. We also thank the IREB Council Shepherds of this handbook, Xavier Franch and Frank Houdek, for their feedback and support. Stefan Sturm provided encouragement and logistic support. We also thank our spouses and families for their patience and support.

Martin Glinz, Hans van Loenhoud, Stefan Staal, and Stan Bühne

November 2020

Understanding the Text Boxes in this Handbook

The handbook includes four differently colored text boxes that complement the explanatory text.

These are:

Definition (corresponding to the Glossary [Glin2020])

Hint

Example

Expression

Version History

Version	Date	Comment	Authors
1.0.0	November 11, 2020	First release	Martin Glinz Hans van Loenhoud Stefan Staal Stan Bühne
1.0.1	December 2020	Minor update	
1.1.0	September 1, 2022	Editorial updates and alignment to the updated CPRE Foundation Level Syllabus v 3.1.0.	Martin Glinz Hans van Loenhoud Stefan Staal Stan Bühne Wim Decoutre
1.1.1	January 1, 2024	New Corporate Design implemented	Stefan Sturm
1.2.0	May 15, 2024	Missing merge nodes added to figure 3.11	Stefan Sturm

Content

1 Introduction and Overview

In this chapter, you will learn what Requirements Engineering (RE) is all about and the value that RE brings.

1.1 Requirements Engineering: What

Since the beginning of human evolution, humans have been building technical and organizational systems to *support* them in completing tasks or achieving objectives. With the rise of engineering, humans have also started to build systems that *automate* human tasks.

Whenever humans decide to build a system to support or automate human tasks, they have to figure out what to build. This means that they have to learn about the desires and needs of the persons or organizations who will use the system, benefit from it, or be impacted by it. In other words, they need to know about the requirements for that system. Requirements form the basis for any development or evolution of systems or parts thereof. Requirements always exist, even when they are not explicitly captured and documented.

The term *requirement* denotes three concepts [Glin2020]:

> **Definition 1.1. Requirement:**
> 1. A need perceived by a *stakeholder*.
> 2. A capability or property that a *system* shall have.
> 3. A documented representation of a need, capability, or property.

A systematically represented collection of requirements—typically for a system—that satisfies given criteria is called a *requirements specification*.

We distinguish between three kinds of requirements:

- *Functional requirements* concern a result or behavior that shall be provided by a function of a system. This includes requirements for data or the interaction of a system with its environment.
- *Quality requirements* pertain to quality concerns that are not covered by functional requirements — for example, performance, availability, security, or reliability.
- *Constraints* are requirements that limit the solution space beyond what is necessary to meet the given functional requirements and quality requirements.

Note that dealing with requirements for projects or development processes is outside the scope of this handbook.

Distinguishing between functional requirements, quality requirements, and constraints is not always straightforward. One proven way to differentiate between them is to ask for the *concern* that a requirement addresses: if the concern is about required results, behavior, or interactions, we have a functional requirement. If it is a quality concern that is not covered by

the functional requirements, we have a quality requirement. If the concern is about restricting the solution space but is neither a functional nor a quality requirement, we have a constraint. The popular rule "*What* the system shall do → functional requirement vs. *how* the system shall do it → quality requirement" frequently leads to misclassifications, particularly when requirements are specified in great detail or when quality requirements are very important.

For example, the requirement "The customer entry form shall contain fields for the customer's name and first name, taking up to 32 characters per field, displaying at least 24 characters, left-bound, with a 12 pt. sanserif font" is a functional requirement even though it contains a lot of information about *how*. As another example, consider a system that processes the measurement data produced by the detector of a high-energy particle accelerator. Such detectors produce enormous quantities of data in real time. If you ask a physicist "What shall the system do?", one of the first answers would probably be that the system must be able to cope with the volume of data produced. However, requirements concerning data volume or processing speed are quality requirements [Glin2007] and not functional requirements.

When people take a systematic and disciplined approach to the specification and management of requirements, we call this *Requirements Engineering (RE)*. The following definition of Requirements Engineering also reflects why we perform RE.

> **Definition 1.2. Requirements Engineering (RE):**
> The systematic and disciplined approach to the specification and management of requirements with the goal of *understanding the stakeholders' desires and needs* and *minimizing the risk* of delivering a system that does not meet these desires and needs.

The concept of *stakeholders* [GlWi2007] is a fundamental principle of Requirements Engineering (see Chapter 2).

> **Definition 1.3. Stakeholder:**
> A person or organization who influences a system's requirements or who is impacted by that system.

Note that influence can also be indirect. For example, some stakeholders may have to follow instructions issued by their managers or organizations.

Following the definition in the CPRE RE glossary [Glin2020], we use the term *system* in a broad sense in this handbook:

> **Definition 1.4. System:**
> 1. In general: a principle for ordering and structuring.
> 2. In engineering: a coherent, delimitable set of elements that—by coordinated action—achieve some purpose.

Note that a system may comprise other systems or *components* as subsystems. The purposes achieved by a system may be delivered by:

- *Deploying* the system at the place(s) where it is used
- Selling/providing the system to its users as a *product*
- Having providers who offer the system's capabilities to users as *services*

We therefore use the term system as an umbrella term which includes products, services, apps, or devices.

1.2 Requirements Engineering: Why

Developing systems (building new ones as well as evolving existing ones) is an expensive endeavor and constitutes a high risk for all participants. At the same time, systems that have practical relevance are too large for a single person to grasp intellectually. Therefore, engineers have developed various principles and practices for handling the risk when developing a system and for mastering the intellectual complexity. Requirements Engineering provides the principles and practices for the requirements perspective.

Adequate Requirements Engineering (RE) adds *value* [Glin2016], [Glin2008] to the process of developing a system:

- RE minimizes the risk of failure or costly modifications in later development stages. The early detection and correction of wrong or missing requirements is much cheaper than the correction of errors and rework caused by missing or wrong requirements in later development stages or even after deployment of a system.
- RE eases the intellectual complexity involved in understanding the problem that a system is supposed to solve and reflecting on potential solutions.
- RE provides a proper basis for estimating development effort and cost.
- RE is a prerequisite for testing the system properly.

Typical symptoms of inadequate RE are missing, unclear, or wrong requirements due to:

- Development teams rushing right into implementing a system due to schedule pressure
- Communication problems between parties involved—in particular, between stakeholders and system developers and among the stakeholders themselves
- The assumption that the requirements are self-evident, which is wrong in most cases
- People conducting RE activities without having adequate education and skills

1.3 Requirements Engineering: Where

Requirements Engineering can be applied to requirements for any kind of system. However, the dominant application case for RE today involves systems in which software plays a major role. Such systems consist of software components, physical elements (technical products, computing hardware, devices, sensors, etc.), and organizational elements (persons, positions, business processes, legal and compliance issues, etc.).

Systems that contain both software and physical components are called *cyber-physical systems*.

Systems that span software, hardware, people, and organizational aspects are called *socio-technical systems*.

Depending on the perspective taken, requirements occur in various forms:

System requirements describe how a system shall work and behave—as observed at the interface between the system and its environment—so that the system satisfies its stakeholders' desires and needs. In the case of pure software systems, we speak of software requirements.

Stakeholder requirements express stakeholders' desires and needs that shall be satisfied by building a system, seen from the stakeholders' perspective.

User requirements are a subset of the stakeholder requirements. They cover the desires and needs of the users of a system.

Domain requirements specify required domain properties of a socio-technical or cyber-physical system.

Business requirements focus on the business goals, objectives, and needs of an organization that shall be achieved by employing a system (or a collection of systems).

The forms of occurrence as presented above match those defined in the standard [ISO29148], with the exception of domain requirements. Due to their importance, we treat domain requirements as a form of their own. The role and importance of domain requirements are discussed in Section 2.2, Principle 4.

1.4 Requirements Engineering: How

The major tasks in RE are the elicitation (Chapter 4), documentation (Chapter 3), validation (Section 4.4), and management (Chapter 6) of requirements. Tool support (Chapter 7) can help perform these tasks. Requirements analysis and requirements conflict resolution are considered to be part of elicitation.

However, there is no universal process that describes when and how RE should be performed when developing a system. For every system development that needs RE activities, a suitable RE process must be tailored from a broad range of possibilities. Factors that influence this tailoring include, for example:

- The overall system development process—in particular, linear and plan-driven vs. iterative and agile
- The development context—in particular, the relationship between the supplier, the customer(s), and the users of a system
- The availability and capability of the stakeholders

There is also a mutual dependency between the requirements work products to be produced (see Section 3.1) and the RE process to be chosen. More details are given in Chapter 5.

1.5 The Role and Tasks of a Requirements Engineer

In practice, very few people have the job title *Requirements Engineer*. We consider people to act in the *role* of a Requirements Engineer when they:

- Elicit, document, validate, and/or manage requirements as part of their duties
- Have in-depth knowledge of RE, which enables them to define RE processes, select appropriate RE practices, and apply these practices properly
- Are able to bridge the gap between the problem and potential solutions

The role of Requirements Engineer is part of several job functions defined by organizations. For example, business analysts, application specialists, product owners, systems engineers, and even developers may act in the role of a Requirements Engineer. Having RE knowledge and skills is also useful for many other professionals—for example, designers, testers, system architects, or CTOs.

1.6 What to Learn about Requirements Engineering

The set of skills that a Requirements Engineer must learn consists of various elements. The foundational elements are covered in the subsequent chapters of this handbook.

Beyond technical and analytical skills, a Requirements Engineer also needs what are referred to as soft skills: the ability to listen, moderate, negotiate, and mediate, as well as empathy for stakeholders and openness to the needs and ideas of others.

RE is governed by a set of fundamental principles that apply to all tasks, activities, and practices of RE. These principles are presented in Chapter 2.

Requirements can be documented in various forms. Various work products can be created at different levels of maturity and detail, from rather informal and temporary ones to very detailed and structured work products that adhere to strict representation rules. It is important to select work products and forms of documentation that are adequate for the situation at hand and to create the chosen work products properly. Work products and documentation practices are presented in Chapter 3.

Requirements can be elaborated (i.e., elicited and validated) with various practices. A Requirements Engineer must be able to select the practices that are best suited in a given situation and apply these practices properly. Elaboration practices are presented in Chapter 4.

Understanding possible processes and working structures enables Requirements Engineers to define a way of working that fits with the specific needs of the system development situation at hand. Processes and working structures are presented in Chapter 5.

Existing requirements can be managed with various practices. Requirements Engineers should be able to understand which requirements management practices support them for which tasks. Management practices are presented in Chapter 6.

Tools make RE more efficient. Requirements Engineers need to know how RE tools can support them and how to select a suitable tool for their situation. Tool support is discussed briefly in Chapter 7.

1.7 Further Reading

The RE terminology used in this handbook is defined in the CPRE Glossary of Requirements Engineering Terminology [Glin2020]. Glinz and Wieringa [GlWi2007] explain the notion of stakeholders. Lawrence, Wiegers, and Ebert [LaWE2001] briefly discuss the risks and pitfalls of RE.

Gause and Weinberg [GaWe1989] wrote one of the first textbooks on RE, which is still worth looking at. Pohl [Pohl2010], Robertson and Robertson [RoRo2012] and Wiegers and Beatty [WiBe2013] are popular textbooks on RE. The course notes of Glinz [Glin2019] provide a slide-based introduction to RE. The textbook by van Lamsweerde [vLam2009] presents a goal-oriented approach to RE. Jackson [Jack1995] contributes an insightful collection of essays about software requirements.

Please be aware that the official textbook for the IREB CPRE Foundation Level version 2.2 [PoRu2015] is no longer fully aligned with version 3.0 of the CPRE Foundation Level Syllabus, on which this handbook is based. However, this textbook still provides a concise introduction to RE and will be updated soon.

There are also textbooks in languages other than English. For example, Badreau and Boulanger [BaBo2014] have written an RE textbook in French. The books by Ebert [Eber2014] and Rupp [Rupp2014] are popular RE textbooks written in German.

2 Fundamental Principles of Requirements Engineering

In this chapter, you will learn about nine basic principles of Requirements Engineering (RE).

2.1 Overview of Principles

RE is governed by a set of fundamental principles that apply to all tasks, activities, and practices in RE. A *task* is a coherent chunk of work to be done (for example, eliciting requirements). An *activity* is an action or a set of actions that a person or group performs to accomplish a task (for example, identifying stakeholders when eliciting requirements). A *practice* is a proven way of how to carry out certain types of tasks or activities (for example, using interviews to elicit requirements from stakeholders).

The principles listed in Table 2.1 form the basis for the practices presented in the subsequent chapters of this handbook.

Table 2.1 Nine fundamental principles of Requirements Engineering

1. *Value orientation*: Requirements are a means to an end, not an end in itself
2. *Stakeholders*: RE is about satisfying the stakeholders' desires and needs
3. *Shared understanding*: Successful systems development is impossible without a common basis
4. *Context*: Systems cannot be understood in isolation
5. *Problem, requirement, solution*: An inevitably intertwined triple
6. *Validation*: Non-validated requirements are useless
7. *Evolution*: Changing requirements are no accident, but the normal case
8. *Innovation*: More of the same is not enough
9. *Systematic and disciplined work*: We can't do without in RE

2.2 The Principles Explained

2.2.1 Principle 1 – Value orientation: Requirements are a means to an end, not an end in itself

The act of writing requirements is not a goal by itself. Requirements are useful—and the effort invested in Requirements Engineering is justified—only if they add *value*

[Glin2016], [Glin2008], cf. Section 1.2. We define the value of a requirement as being its *benefit* minus its *cost*. The benefit of a requirement is the degree to which it contributes to building successful systems (that is, systems that satisfy the desires and needs of their stakeholders) and to reducing the risk of failure and costly rework in system development. The cost of a requirement amounts to the cost for eliciting, validating, documenting, and managing it.

Reducing the risk of rework during development is a constituent part of the benefit of a well-crafted requirement. Detecting and fixing a missed or wrong requirement during implementation or when the system is already in operation can easily cost one or two orders of magnitude more than specifying that requirement properly right from the beginning. Consequently, a significant amount of the benefit of requirements comes from costs saved during the implementation and operation of a system.

In other words, the benefits of RE are often long-term benefits, whereas the costs are immediate. This must be kept in mind when setting up a new project. Reducing costs in the short term by spending less for RE has a price: it considerably increases the risk of expensive rework in later stages of the project.

The *value of Requirements Engineering* can be considered to be the cumulative value of the requirements specified. As customers typically pay for systems to be implemented, but not for the requirements needed to do that, the economic value of RE is mostly an indirect one. This effect is reinforced by the fact that the benefit of requirements that stem from reduced rework costs is an indirect one: it saves costs during implementation and operation.

> The economic effects of Requirements Engineering are mostly indirect ones; RE as such just costs.

To optimize the value of a requirement, Requirements Engineers have to strike a proper balance between the benefit and the cost of a requirement. For example, eliciting and documenting a stakeholder's need as a requirement eases the communication of this need among all parties involved. This increases the probability that the system to be built will eventually satisfy this need, which constitutes a benefit. The less ambiguously and the more precisely the requirement is stated, the higher its benefit, because this reduces the risk of costly rework due to misinterpretation of the requirements by the system architects and development teams. On the other hand, increasing the degree of unambiguity and precision of a requirement also increases the cost involved in eliciting and documenting the requirement.

Actually, the amount of RE required to achieve requirements with optimal value depends on numerous factors given by the specific situation in which requirements are being created and used. Obviously, the risk of building a system that eventually does not satisfy the desires and needs of its stakeholders, which may result in failure or costly rework, is *the driving force* that determines the amount of RE required. First and foremost, the criticality of every

requirement should be assessed in terms of the importance of the stakeholder(s) who state the requirement (see Principle 2) and the impact of missing the requirement (Figure 2.1).

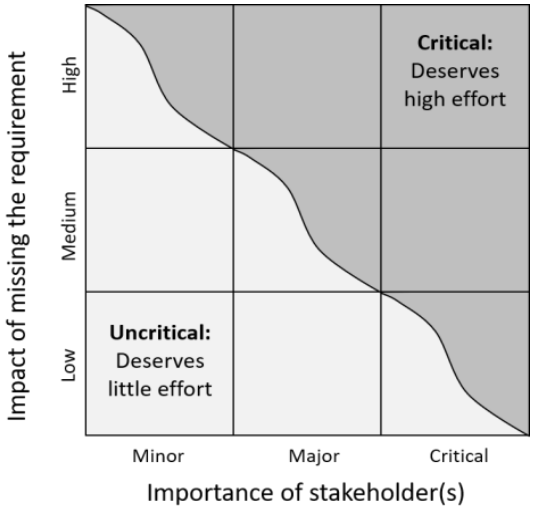


Figure 2.1 Assessing the criticality of a requirement [Glin2008]

In addition, the following influencing factors should be considered:

- Effort needed to specify the requirement
- Distinctiveness of the requirement (how much it contributes to the success of the overall system)
- Degree of shared understanding between stakeholders and developers and among stakeholders
- Existence of reference systems (that can serve as a specification by example)
- Length of feedback cycle (the time between getting a requirement wrong and detecting the error)
- Kind of customer–supplier relationship
- Regulatory compliance required

We summarize this issue in two rules of thumb:

- The optimal amount of RE to be invested depends on the specific situation and is determined by many influencing factors.
- The effort invested into RE should be inversely proportional to the risk you are willing to take.

2.2.2 Principle 2 – Stakeholders: RE is about satisfying the stakeholders' desires and needs

The eventual goal of building a system is that the system, when it is used, solves problems that its users need to solve and satisfies the expectations of further people—for example, those who have ordered and paid for the system, or those who are responsible for security in the organization that uses the system. Therefore, we have to figure out the needs and

expectations of the people who have a stake in the system, the system's *stakeholders* [GlWi2007]. The core goals of RE are *understanding the stakeholders' desires and needs* and *minimizing the risk* of delivering a system that does not meet these desires and needs; see Definition 1.2 in Section 1.2.

Every stakeholder has a *role* in the context of the system to be built—for example, user, client, customer, operator, or regulator. Depending on the RE process used, the developers of a system can also be stakeholders. This is frequently the case in agile and in market-oriented development. A stakeholder may also have more than one role at the same time. For every relevant stakeholder role, suitable people acting in this role must be selected as representatives.

For stakeholder roles with too many individuals or when individuals are unknown, *personas* (fictitious characters that represent a group of users with similar characteristics) can be defined as a substitute. For systems that are already in use, users who provide feedback about the system or ask for new features should also be considered as stakeholders.

It makes sense to classify the stakeholders into three categories with respect to the degree of influence that a stakeholder has on the success of the system:

- *Critical:* not considering these stakeholders will result in severe problems and probably make the system fail or render it useless.
- *Major:* not considering these stakeholders will have an adverse impact on the success of the system but not make it fail.
- *Minor:* not considering these stakeholders will have no or minor influence on the success of the system.

This classification is helpful when assessing the criticality of a requirement (see Figure 2.1) and when negotiating conflicts between stakeholders (see below).

It is not sufficient to consider only the requirements of end users and customers. Doing this would mean that we might miss critical requirements from other stakeholders, which can easily lead to development projects that fail or overrun their budgets and deadlines.

Involving the right people in the relevant stakeholder roles is crucial for successful RE.

Practices for identifying, prioritizing, and working with stakeholders are discussed in Chapter 4.

Stakeholders in different roles naturally have different viewpoints [NuKF2003] of a system to be developed. For example, users typically want a system to support their tasks in an optimal way, the managers who order the system want to get it at a reasonable cost, and the organization's chief security officer cares primarily about the security of the system. Even stakeholders in the same role may have different needs. For example, in the group of end users, casual users have user interface requirements that may differ strongly from those of professional users.

As a consequence, it is not sufficient to just collect requirements from stakeholders. It is vital to identify inconsistencies and conflicts between the requirements of different stakeholders and to resolve these, be it by finding a consensus, by overruling, or by specifying system variants for stakeholders who factually have different needs; see Section 4.3.

2.2.3 Principle 3 – Shared understanding: Successful systems development is impossible without a common basis

System development, including RE, is a multi-person endeavor. To make such an endeavor a success, the people involved need a *shared understanding* of the problem and the requirements that stem from it [GlFr2015].

RE creates, fosters, and secures shared understanding between and among the parties involved: stakeholders, Requirements Engineers, and developers. We distinguish between two forms of shared understanding:

- *Explicit shared* understanding is achieved through carefully elicited, documented, and agreed requirements. This is the primary goal of RE in plan-driven processes.
- *Implicit shared* understanding is based on shared knowledge about needs, visions, context, etc. In agile RE, when requirements are not fully specified in writing, reliance on implicit shared understanding is key.

Both implicit and explicit shared understanding may be *false*, meaning that people believe that they have a shared understanding of an issue but in fact interpret this issue in different ways. Therefore, we can never rely blindly on shared understanding. Instead, the task of RE is to create and foster shared understanding and also secure it—that is, assess whether there is a true shared understanding. To limit the effort involved, it is vital to concentrate on shared understanding about *relevant* things—that is, those aspects that lie within the context boundary of a system (cf. Principle 4).

Even with a perfect shared understanding, important requirements may still be missed because nobody considered them. Figure 2.2 illustrates different situations of shared understanding with a simple example of a couple that wants to install a swing in their garden for their children [Glin2019]. The sticky note in the middle symbolizes a written specification.

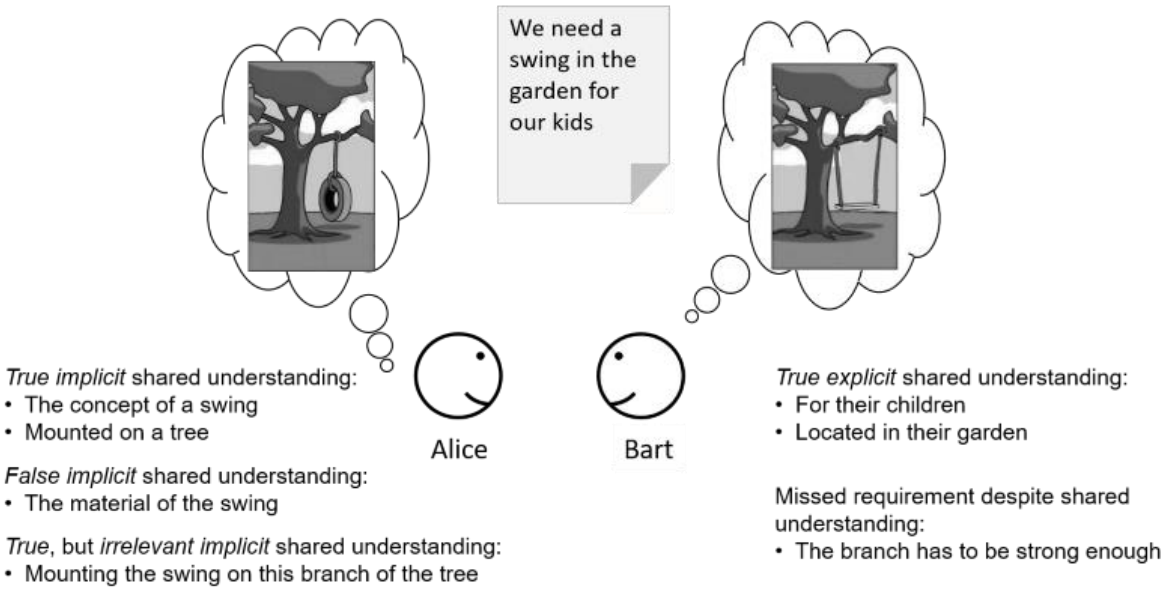


Figure 2.2 Different situations of shared understanding—illustrated with an example of a couple that wants to install a swing for their children

Proven practices for *achieving* shared understanding include working with glossaries (Section 3.5), creating prototypes (Section 0), or using an existing system as a reference point.

The main means for *assessing* true explicit shared understanding in RE is thoroughly validating all specified requirements (cf. Principle 6 and Section 4.4). Practices for assessing implicit shared understanding include providing examples of expected outcomes, building prototypes, or estimating the cost of implementing a requirement. The most important practice for reducing the impact of false shared understanding is using a process with short feedback loops (Chapter 5).

There are factors that constitute enablers or obstacles of shared understanding. For example, enablers are:

- Domain knowledge
- Domain-specific standards
- Previous successful collaboration
- Existence of reference systems known by all people involved
- Shared culture and values
- Informed (not blind!) mutual trust

Obstacles are:

- Geographic distance
- Supplier–customer relationship guided by mutual distrust
- Outsourcing
- Regulatory constraints
- Large and diverse teams
- High turnover among the people involved

The lower the probability and impact of false shared understanding and the better the ratio between enablers and obstacles, the more RE can rely on implicit shared understanding. Conversely, the fewer enablers and the more obstacles to shared understanding we have and the higher the risk and impact of false shared under-standing for a requirement, the more such requirements have to be specified and validated explicitly.

2.2.4 Principle 4 - Context: Systems cannot be understood in isolation

Requirements never come in isolation. They refer to *systems* that are embedded in a *context*. While the term *context* in general denotes the network of thoughts and meanings needed for understanding phenomena or utterances, it has a special meaning in RE.

> Definition 2.1. Context (in RE):
> The part of a system's environment being relevant for understanding the system and its requirements.

The context of a system is delimited by the system boundary and the context boundary [Pohl2010] (see Figure 2.3).

> Definition 2.2. Context boundary:
> The boundary between the context of a system and those parts of the application domain that are irrelevant for the system and its requirements.

The *context boundary* separates the relevant part of the environment of a system to be developed from the irrelevant part—that is, the part that does not influence the system to be developed and, thus, does not have to be considered during Requirements Engineering.

> Definition 2.3. System boundary:
> The boundary between a system and its surrounding context.

The *system boundary* delimits the system as it shall be after its implementation and deployment. The system boundary is often not clear initially and it may change over time. Clarifying the system boundary and defining the external interfaces between a system and the elements in its context are genuine RE tasks.

The system boundary frequently coincides with the *scope* of a system development.

> **Definition 2.4. Scope:**
> The range of things that can be shaped and designed when developing a system.

Sometimes, however, the system boundary and its scope do not match (see Figure 2.3). There may be components within the system boundary that have to be reused as they are (i.e., they cannot be shaped or designed), which means that they are out of scope. On the other hand, there may be things in the system context that can be re-designed when the system is developed, which means that they are in scope.

As the external interfaces reside at the system boundary, RE must determine which of these interfaces are in scope (that means, they can be shaped and designed in the development process) and which ones are given and out of scope.

It is not sufficient to consider just the requirements within the system boundary.

First, when the scope includes parts of the system context, as shown in Figure 2.3, context changes within the scope may impact the system's requirements. For example, when a business process shall be partially automated by a system, it may be useful to adapt the process in order to simplify its automation. Obviously, such adaptation impacts the requirements of the system.

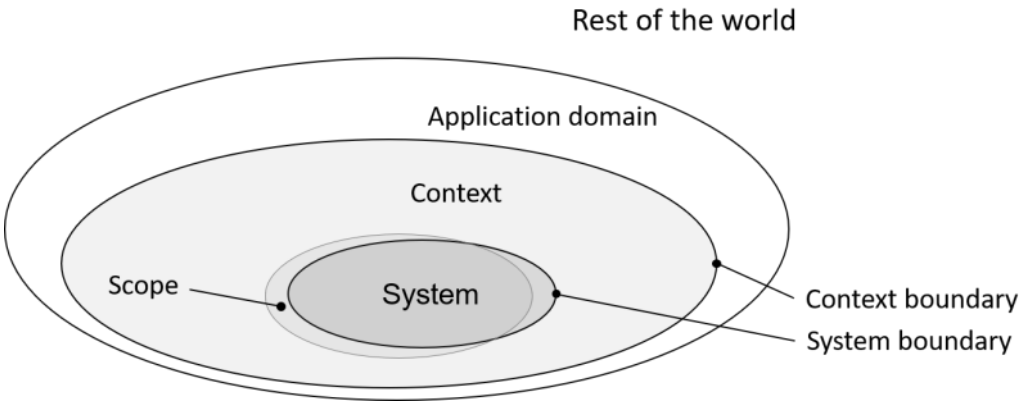


Figure 2.3 System, context, and scope

Second, there may be real-world phenomena in the system context that a system shall monitor or control. Requirements for such phenomena must be stated as domain requirements and must be adequately mapped to system requirements. For example, in a car equipped with an automatic gearbox, there is a requirement that the parking position can be engaged only when the car is not moving. In the context of a software system that controls the gearbox, this is a domain requirement. In order to satisfy this requirement, the controller needs to know whether or not the car is moving. However, the controller cannot sense this phenomenon directly. Hence, the real-world phenomenon "car is not moving" must be mapped to a phenomenon that the control system can sense—for example, input

from a sensor that creates pulses when a wheel of the car is spinning. The domain requirement concerning engaging the parking position is then mapped to a system requirement such as "The gearbox control system shall enable the engagement of the parking position only if no pulses are received from the wheel spinning sensors."

Third, there may be requirements that cannot be satisfied by any system implementation unless certain *domain requirements* and *domain assumptions* in the context of the system hold. Domain assumptions are assumptions about real-world phenomena in the context of a system. For example, consider an air traffic control system (ATS). The requirement "R1: The ATS shall maintain accurate positions for all aircraft controlled by the system" is an important system requirement. However, this requirement can be met only if the radar in the context of the ATS satisfies the requirements of correctly identifying all aircraft in the airspace controlled by the radar and correctly determining their position. In turn, these requirements can be satisfied only if all aircraft spotted by the radar respond properly to the interrogation signals sent by the radar.

Furthermore, requirement R1 can be met only if certain domain assumptions in the context of the ATS hold—for example, that the radar is not jammed by a malicious attacker and that no aircraft are flying at an altitude that is lower than the radar can detect.

RE goes beyond considering the requirements within the system boundary and defining the external interfaces at the system boundary. RE must also deal with phenomena in the system context.

Consequently, RE must also consider issues in the system context:

- If changes in the context may occur, how do they impact the requirements for the system?
- Which requirements in the real-world context are relevant for the system to be developed?
- How can such real-world requirements be mapped adequately to requirements for the system?
- Which assumptions about the context must hold such that the system will work properly and the requirements in the real world will be met?

2.2.5 Principle 5 – Problem, requirement, solution: An inevitably intertwined triple

Problems, their solutions, and requirements are closely and inevitably intertwined [SwBa1982]. Every situation in which people are not satisfied with the way they are doing things can be considered as the occurrence of a *problem*. In order to eliminate that problem, a socio-technical system may be developed and deployed. *Requirements* for that system must be captured in order to make the system an effective *solution* to the problem. Specifying requirements does not make sense if there is no problem to solve or if no solution

will be developed. Neither does it make sense to develop a solution that is searching for a problem to solve or for requirements to satisfy.

It is important to note that problems, requirements, and solutions do not necessarily occur in this order. For example, when designing an innovative system, solution ideas create user needs that have to be worked out as requirements and implemented in an actual solution.

Problems, requirements, and solutions can be intertwined in many ways:

- Hierarchical intertwinement: when developing large systems with a multi-level hierarchy of subsystems and components, high-level requirements lead to high-level design decisions, which in turn inform lower-level requirements that lead to lower-level design decisions, etc.
- Technical feasibility: specifying non-feasible requirements is a waste of effort; however, it may only be possible to assess the feasibility of a requirement when exploring technical solutions.
- Validation: prototypes, which are a powerful means for validating requirements, constitute partial solutions of the problem.
- Solution bias: different stakeholders may envisage different solutions for a given problem, with the consequence that they specify different, conflicting requirements for that problem.

The intertwinement of problems, requirements, and solutions also has consequences for the development process for a system:

- Strictly separating RE from system design and implementation activities is rarely possible. Therefore, strict waterfall development processes do not work well.
- Nevertheless, Requirements Engineers aim to separate problems, requirements, and solutions from each other as far as possible when thinking, communicating, and documenting. This *separation of concerns* makes RE tasks easier to handle.

Despite the inevitable intertwinement of problems, requirements, and solutions, Requirements Engineers strive to separate requirements concerns from solution concerns when thinking, communicating, and documenting.

2.2.6 Principle 6 – Validation: Non-validated requirements are useless

When a system is developed, the final system deployed shall satisfy the stakeholders' desires and needs. However, performing this check at the very end of development is very risky. In order to control the risk of unsatisfied stakeholders from the beginning, validation of requirements must start during RE (see Figure 2.4).

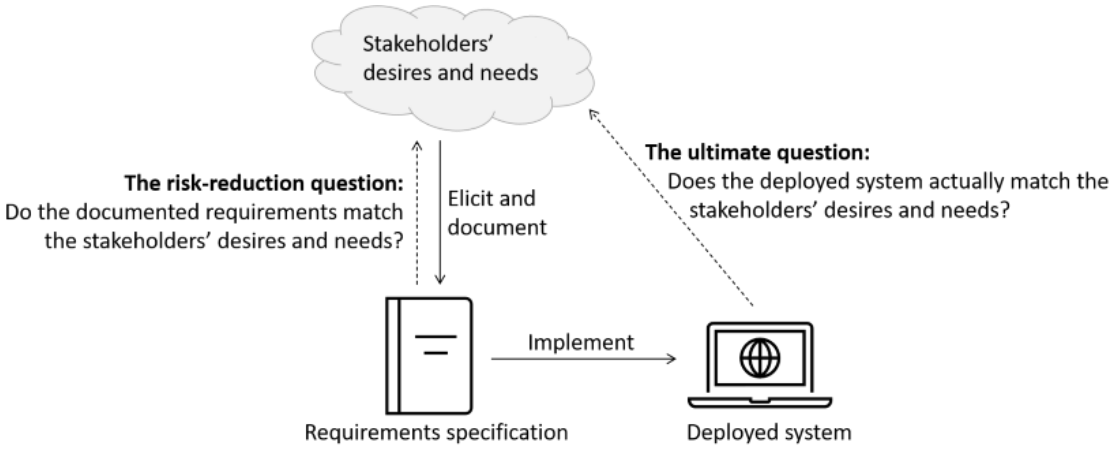


Figure 2.4 Validation [Glin2019]

> **Definition 2.5. Validation:**
> The process of confirming that an item (a system, a work product, or a part thereof) matches its stakeholders' needs.

In RE, *validation* is the process of confirming that the documented requirements match the stakeholders' needs; in other words, confirming whether the right requirements have been specified.

> Validation is a core activity in RE: there is no specification without validation.

When validating requirements, we have to check whether:

- Agreement about the requirements has been achieved among the stakeholders (conflicts resolved, priorities set)
- The stakeholders' desires and needs are adequately covered by the requirements
- The domain assumptions (see Principle 4 above) are reasonable—that is, we can expect that these assumptions can be met when the system is deployed and operated

Practices for validating requirements are discussed in Section 4.4.

2.2.7 Principle 7 – Evolution: Changing requirements are no accident, but the normal case

Every technical system is subject to evolution. Needs, businesses, and capabilities change continuously. As a natural consequence, the requirements for systems that are expected to satisfy needs, support businesses, and use technical capabilities will also change. Otherwise, such systems and their requirements progressively lose their value and eventually become useless.

A requirement may change while Requirements Engineers are still eliciting other requirements, when the system is under implementation, or when it is deployed and being used.

There are many reasons that lead to requests to change a requirement or a set of requirements for a system, for example:

- Changed business processes
- Competitors launching new products or services
- Clients changing their priorities or opinions
- Changes in technology
- Feedback from system users asking for new or changed features
- Detection of errors in requirements or detection of faulty domain assumptions

Requirements may also change due to feedback from stakeholders when validating requirements, due to the detection of faults in previously elicited requirements, or due to changed needs.

As a consequence, Requirements Engineers must pursue two seemingly contradictory goals:

- Permit requirements to change, because trying to ignore the evolution of requirements would be futile.
- Keep requirements stable, because without some stability in the requirements, the cost for change can become prohibitively high. Also, development teams cannot develop systematically if requirements change on a daily basis.

> Requirements Engineers need to manage the evolution of requirements. Otherwise, the evolution will manage them.

Change processes for requirements that address both goals are discussed in Section 6.7.

2.2.8 Principle 8 – Innovation: More of the same is not enough

While RE is concerned with satisfying the stakeholders' desires and needs, Requirements Engineers who just play the role of the stakeholders' voice recorder, specifying exactly what the stakeholders tell them, are doing the wrong job. Giving stakeholders exactly what they want means missing out on the opportunity of doing things better than before.

For example, imagine the following scenario. An insurance company wants to renew the reporting system for its agents. The most frequently used report is a table with 18 columns, which is about twice as wide as the screen when displayed on the agents' laptop computers. Viewing this report thus requires a lot of scrolling. The stakeholders therefore want to be able to zoom in the report, using plus and minus buttons on the screen. In this situation, good Requirements Engineers will not just record this as a requirement. Instead, they will start to ask questions. It turns out that the company is going to replace the agents' laptops with tablets. Hence, implementing two-finger gestures instead of the required buttons will make zooming much easier. Furthermore, it turns out that three columns in the report can be eliminated with a slight change to the reporting rules, which the company agrees to make. Also, only six columns of the report are always needed; the remaining columns are used only in special cases.

Taking this into account, the Requirements Engineers would suggest that the stakeholders require that (1) the report shall show the same information as in the current system, minus the content of the three eliminated columns; (2) when the report is opened, only the six important columns are displayed in full width, while the other columns are collapsed to minimal width; and (3) that agents can expand a collapsed column by tapping its header (and collapse it again with another tap).

This way, the agents will get a system that does not simply add a workaround for viewing an oversized report. Instead, the system will solve the agents' problem with an innovative feature for filtering information and will also feature an intuitive means of zooming.

This is how innovation emerges. Good Requirements Engineers are innovation-aware: they strive not just to satisfy stakeholders but to make them happy, excited, or feel safe [KSTT1984]. At the same time, they avoid the trap of believing that they know everything better than the stakeholders do.

Good Requirements Engineers go beyond what their stakeholders tell them.

On a small scale, RE shapes innovative systems by striving for exciting new features and ease of use. Beyond that, Requirements Engineers also need to look for the big picture, exploring with the stakeholders whether there are any disruptive ways of doing things, leading to large-scale innovation [MaGR2004].

Section 4.2 discusses several techniques for fostering innovation in RE.

2.2.9 Principle 9 - Systematic and disciplined work: We can't do without in RE

RE is not an art but a discipline, which calls for RE to be performed in a systematic and disciplined way. Regardless of the process(es) used to develop a system, we need to employ suitable RE processes and practices for systematically eliciting, documenting, validating, and managing requirements. Even when a system is developed in an ad hoc fashion, a systematic and disciplined approach to RE (for example, by systematically fostering shared understanding, see Principle 3) will improve the quality of the resulting system.

Agility and flexibility are not valid excuses for an unsystematic, ad hoc style of work in RE.

However, there is neither a universal RE process nor a universal set of RE practices that work well in every given situation or at least in most situations: there is no "one size fits all" in RE.

Systematic and disciplined work means that Requirements Engineers:

- Configure an RE process that is well suited for the problem at hand and fits well with the process used for developing the system (see Chapter 5).
- From the set of RE practices and work products available, select those that are best suited for the given problem, context, and working environment (see Chapters 3, 4 and 6).
- Do not always use the same process, practices, and work products.
- Do not reuse processes and practices from past successful RE work without reflection.

2.3 Further Reading

Glinz [Glin2008] discusses the value of quality requirements and of requirements in general [Glin2016].

Glinz and Wieringa [GlWi2007] explain the notion and importance of stakeholders.

Glinz and Fricker [GlFr2015] discuss the role and importance of shared understanding.

The papers by Jackson [Jack1995b] and Gunter et al. [GGJZ2000] are fundamental for the problem of requirements in context. The role of context in RE is also discussed by Pohl [Pohl2010].

Gause and Weinberg [GaWe1989] discuss the interdependence of problems and solutions. Swartout and Balzer [SwBa1982] were the first to point out that creating a complete specification before starting implementation is rarely possible.

Validation is covered in any RE textbook. Grünbacher and Seyff [GrSe2005] discuss how to achieve agreement by negotiating about requirements.

Kano et al. [KSTT1984] were among the first to stress the role of innovation. Maalej, Nayebi, Johann, and Ruhe [MNJR2016] discuss the use of explicit and implicit user feedback for RE. Maiden, Gitzikis, and Robertson [MaGR2004] discuss how creativity can foster innovation in RE. Gorschek et al. [GFPK2010] outline a systematic innovation process.

3 Work Products and Documentation Practices

Traditional Requirements Engineering (RE) calls for the writing of a comprehensive, complete, and unambiguous requirements specification [IEEE830], [Glin2016]. While it is still appropriate to create fully-fledged requirements specifications in many cases, there are also many other cases where the cost of writing such specifications exceeds their benefit. For example, fully-fledged requirements specifications are useful or even necessary when tendering or outsourcing the design and implementation of a system or when a system is safety-critical and regulatory compliance is required. On the other hand, where stakeholders and developers join forces to define and develop a system iteratively, writing a comprehensive requirements specification does not make sense. It is therefore vital in RE to adapt the documentation to the project context and to select work products for documenting requirements and requirements-related information that yield optimal value for the project.

In this chapter, you will learn about the typical RE work products and how to create them.

3.1 Work Products in Requirements Engineering

There are a variety of work products that are used in RE.

> **Definition 3.1. Work product:**
> A recorded intermediate or final result generated in a work process.

We consider the term *artifact* as a synonym for work product. We prefer the term work product over artifact to express the connotation that a work product is the result of work performed in a work process.

According to this definition, an RE work product can be anything that expresses requirements, from a single sentence or diagram to a system requirements specification that covers hundreds of pages. It is also important to note that a work product may contain other work products.

3.1.1 Characteristics of Work Products

Work products can be characterized by the following facets: *purpose*, *size*, *representation*, *lifespan*, and *storage*.

Table 3.1 gives an overview of typical work products used in RE along with their respective purpose (that is, what the work product specifies or provides) and typical size. The table is structured into four groups: work products for single requirements, coherent sets of requirements, documents or documentation structures, and other work products.

There are many different ways to represent a work product. In RE, representations based on *natural language*, *templates,* and *models* are of particular importance. These are discussed in Sections 3.2, 3.3, and 3.4, respectively. There are further representations, such as drawings or prototypes, which are covered in Section3.7.

Every work product has a *lifespan*. This is the period of time from the creation of the work product until the point where the work product is discarded or becomes irrelevant. We distinguish between three categories of work products with respect to lifespan: *temporary*, *evolving*, and *durable* work products.

Temporary work products are created to support communication and create shared understanding (for example, a sketch of a user–system interaction created in a workshop). Temporary work products are discarded after use; no metadata is kept about these work products.

Evolving work products emerge in several iterations over time (for example, a collection of user stories that grows in both the number of stories and the story content). Some metadata (at least the owner, status, and revision history) should be kept for every evolving work product. Depending on the importance and status of a work product, change control procedures need to be applied when modifying an evolving work product.

Durable work products have been baselined or released (for example, a requirements specification that is part of a contract or a sprint backlog that is implemented in a given iteration). A full set of metadata must be kept to manage the work product properly and an elaborate change process must be followed to change a durable work product (Chapter 6).

A temporary work product may become an evolving one when Requirements Engineers decide to keep a work product and develop it further. In this case, some metadata should be added in order to keep the evolution of the work product under control. When an evolving work product is baselined or released, it changes its lifespan status from evolving to durable.

Table 3.1 Overview of RE work products

Work product	Purpose: The work product specifies /provides	Size*
Single requirements		
Individual requirement	A single requirement, typically in textual form	S
User story	A function or behavior from a stakeholder's perspective	S
Coherent sets of requirements		
Use case	A system function from an actor's or user's perspective	S–M

Work product	Purpose: The work product specifies /provides	Size*
Graphic model	Various aspects, for example, context, function, behavior (see Section 3.4)	M
Task description	A task that a system shall perform	S–M
External interface description	The information exchanged between a system and an actor in the system context	M
Epic	A high-level view of a stakeholder need	M
Feature	A distinguishing characteristic of a system	S–M
Documents or documentation structures		
System requirements specification**	A comprehensive requirements document	L–XL
Product and sprint backlog	A list of work items, including requirements	M–L
Story map	A visual arrangement of user stories	M
Vision	A conceptual imagination of a future system	M
Other work products		
Glossary	Unambiguous and agreed common terminology	M
Textual note or graphic sketch	A memo for communication and understanding	S
Prototype	A specification by example, particularly for understanding, validating, and negotiating about requirements	S–L

*: S: Small, M: Medium, L: Large, XL: Very large
**: Other examples are: business requirements specification, domain requirements specification, stakeholder/user requirements specification or software requirements specification

Nowadays, most work products are stored electronically as files, in databases, or in RE tools. Informal, temporary work products may also be stored on other media—for example, paper or sticky notes on a Kanban board.

3.1.2 Abstraction Levels

Requirements and their corresponding work products occur at various abstraction levels—from, for example, high-level requirements for a new business process, down to requirements at a very detailed level, such as the reaction of a specific software component to an exceptional event.

Business requirements, domain requirements, and stakeholder/user requirements typically occur at a higher level of abstraction than system requirements. When a system consists of a hierarchy of subsystems and components, we have system requirements at the corresponding abstraction levels for subsystems and components.

When business requirements and stakeholder requirements are expressed in durable work products—such as business requirements specifications, stakeholder requirements specifications, or vision documents—they precede the specification of system requirements. For example, in contractual situations, where a customer orders the development of a system from a supplier, the customer frequently creates and releases a stakeholder requirements specification. The supplier then uses this as the basis for producing a system requirements specification. In other projects, business requirements, stakeholder requirements, and system requirements may co-evolve.

Some work products, such as individual requirements, sketches, or process models, occur at all levels. Other work products are specifically associated with certain levels. For example, a system requirements specification is associated with the system level. Note that an individual requirement at a high abstraction level may be refined into several detailed requirements at more concrete levels.

The choice of the proper abstraction level particularly depends on the subject to be specified and the purpose of the specification. For example, if the subject to be specified is a low-level part of the problem to be solved, it will be specified at a rather low abstraction level. It is important, however, not to mix requirements that are at different abstraction levels. For example, in the specification of a healthcare information system, when writing a detailed requirement about photos on client ID cards, the subsequent paragraph should not state a general system goal such as reducing healthcare cost while maintaining the current service level for clients. In small and medium-sized work products (for example, user stories or use cases), requirements should be at more or less the same abstraction level. In large work products such as a system requirements specification, requirements at different levels of abstraction should be kept separate by structuring the specification accordingly (Section 3.6).

Requirements naturally occur at different levels of abstraction. Selecting work products that are adequate for a given level of abstraction and properly structuring work products that contain requirements at multiple abstraction levels is helpful.

3.1.3 Level of Detail

When specifying requirements, Requirements Engineers have to decide on the level of detail in which the requirements shall be specified. However, deciding which level of detail is appropriate or even optimal for a given requirement is a challenging task.

For example, in a situation where the customer and the supplier of a system collaborate closely, it might be sufficient to state a requirement about a data entry form as follows: "The system shall provide a form for entering the personal data of the customer." In contrast, in a situation where the design and implementation of the system are outsourced to a supplier with little or no domain knowledge, a detailed specification of the customer entry form will be necessary.

The level of detail to which requirements should be specified depends on several factors, in particular:

- The problem and project context: the harder the problem and the less familiar the Requirements Engineers and developers are with the project context, the more detail is necessary.
- The degree of shared understanding of the problem: when there is low implicit shared understanding (see Principle 3 in Chapter 2), explicit, detailed specifications are required to create the necessary degree of shared understanding.
- The degree of freedom left to designers and programmers: less detailed requirements give the developers more freedom.
- Availability of rapid stakeholder feedback during design and implementation: when rapid feedback is available, less detailed specifications suffice to control the risk of developing the wrong system.
- Cost vs. value of a detailed specification: the higher the benefit of a requirement, the more we can afford to specify it in detail.
- Standards and regulations: Standards imposed and regulatory constraints may mean that requirements have to be specified in more detail than would otherwise be necessary.

There is no universally "right" level of detail for requirements. For every requirement, the adequate level of detail depends on many factors. The greater the level of detail in the requirements specified, the lower the risk of eventually getting something that has unexpected or missing features or properties. However, the cost for the specification increases as the level of detail increases.

3.1.4 Aspects to be Considered

Regardless of the RE work products being used, several aspects need to be considered when specifying requirements [Glin2019].

First, as there are functional requirements, quality requirements, and constraints (see Section 1.1), Requirements Engineers have to make sure that they cover all three kinds of requirements when documenting requirements. In practice, stakeholders tend to omit quality requirements because they take them for granted.

They also tend to specify constraints as functional requirements. It is therefore important that the Requirements Engineers get this right.

When looking at functional requirements, we observe that they pertain to different aspects, as, for example, a required data structure, a required order of actions, or the required reaction to some external event. We distinguish between three major aspects: *structure and data*, *function and flow*, and *state and behavior*.

The *structure and data* aspect focuses on requirements concerning the static structure of a system and the (persistent) data that a system must know in order to perform the required functions and deliver the required results.

The *function and flow* aspect deals with the functions that a system shall provide and the flow of control and data within and between functions for creating the required results from given inputs.

The *state and behavior* aspect concentrates on specifying the state-dependent behavior of a system—in particular, how a system shall react to which external event depending on the system's current state.

When dealing with *quality requirements*, such as usability, reliability, or availability, a quality model—for example, the model provided by ISO/IEC 25010 [ISO25010]—can be used as a checklist.

Within the quality requirements, *performance requirements* are of particular importance. Performance requirements deal with:

- Time (e.g., for performing a task or reacting to external events)
- Volume (e.g., required database size)
- Frequency (e.g., of computing a function or receiving stimuli from sensors)
- Throughput (e.g., data transmission or transaction rates)
- Resource consumption (e.g., CPU, storage, bandwidth, battery)

Some people also consider the required accuracy of a computation as a performance requirement.

Whenever possible, measurable values should be specified. When values follow a probability distribution, specifying just the average does not suffice. If the distribution function and its parameters cannot be specified, Requirements Engineers should strive to specify minimum and maximum values or 95 percent values in addition to the averages.

Documenting quality requirements beyond performance requirements is notoriously difficult.

Qualitative representations, such as "The system shall be secure and easy to use," are ambiguous and thus difficult to achieve and validate.

Quantitative representations are measurable, which is a big asset in terms of systematically achieving and validating a quality requirement. However, they raise principal difficulties (for example, how can we state security in quantitative terms?) and can be quite expensive to specify.

Operationalized representations state a quality requirement in terms of functional requirements for achieving the desired quality. For example, a data security requirement may be expressed in terms of a login function that restricts the access to the data and a function that encrypts the stored data. Operationalized representations make quality requirements testable but may also imply premature design decisions.

The often-heard rule "Only a quantified quality requirement is a good quality requirement" is outdated and may lead to quality requirements having low or even negative value due to the high effort involved in the quantification. Instead, a risk-based approach should be used [Glin2008].

Qualitative representations of quality requirements *suffice* in the following situations:

- There is sufficient implicit shared understanding between stakeholders, Requirements Engineers, and developers.
- Stakeholders, Requirements Engineers, and developers agree on a known solution that satisfies the requirements.
- Stakeholders only want to give general quality directions and trust the developers to get the details right.
- Short feedback loops are in place such that problems can be detected early.

When developers are *able to generalize from examples*, specifying quality requirements in terms of quantified examples or comparisons to an existing system is a cheap and effective way of documenting quality requirements.

Only in cases where there is a *high risk* of not meeting the stakeholders' needs, particularly when quality requirements are *safety-critical*, should a fully quantified representation or an operationalization in terms of functional requirements be considered.

When specifying *constraints*, the following categories of constraints should be considered:

- *Technical*: given interfaces or protocols, components, or frameworks that have to be used, etc.
- *Legal*: restrictions imposed by laws, contracts, standards, or regulations
- *Organizational*: there may be constraints in terms of organizational structures, processes, or policies that must not be changed by the system.
- *Cultural*: user habits and expectations are to some extent shaped by the culture the users live in. This is a particularly important aspect to consider when the users of a

system come from different cultures or when Requirements Engineers and developers are rooted in a different culture to the system's users.

- *Environmental*: when specifying cyber-physical systems, environmental conditions such as temperature, humidity, radiation, or vibration may have to be considered as constraints; energy consumption and heat dissipation may constitute further constraints.
- *Physical*: when a system comprises physical components or interacts with them, the system becomes constrained by the laws of physics and the properties of materials used for the physical components.
- Furthermore, *particular solutions or restrictions demanded by important stakeholders* also constitute constraints.

Finally, requirements can only be understood in *context* (see Principle 4 in Chapter 2). Consequently, a further aspect has to be considered, which we call *context and boundary*.

The *context and boundary* aspect covers domain requirements and domain assumptions in the context of the system, as well as the external actors that the system interacts with and the external interfaces between the system and its environment at the system boundary.

There are many interrelationships and dependencies between the aspects mentioned above. For example, a request issued by a user (context) may be received by the system via an external interface (boundary), trigger a state transition of the system (state and behavior), which initiates an action (function) followed by another action (flow) that requires data with some given structure (structure and data) to provide a result to the user (context) within a given time interval (quality).

Some work products focus on a specific aspect and abstract from the other aspects. This is particularly the case for requirements models (Section 3.4). Other work products, such as a system requirements specification, cover all these aspects. When different aspects are documented in separate work products or in separate chapters of the same work product, these work products or chapters must be kept consistent with each other.

Many different aspects need to be considered when documenting requirements, in particular, functionality (structure and data, function and flow, state and behavior), quality, constraints, and surrounding context (context and boundary).

3.1.5 General Documentation Guidelines

Independently of the techniques used, there are some general guidelines that should be followed when creating RE work products:

- Select a work product type that fits the intended purpose.
- Avoid redundancy by referencing content instead of repeating the same content again.
- Avoid inconsistencies between work products, particularly when they cover different aspects.

- Use terms consistently, as defined in the glossary.
- Structure work products appropriately—for example, by using standard structures.

3.1.6 Work Product Planning

Each project setting and each domain is different, so the set of resulting work products must be defined for each endeavor. The parties involved, particularly the Requirements Engineers, stakeholders, and project/product owners or managers need to agree upon the following issues:

- In which work products shall the requirements be recorded and for what purpose (see Table 3.1)?
- Which abstraction levels need to be considered (Section 3.1.2)?
- Up to which level of detail must requirements be documented at each abstraction level (Section 03.1.3)?
- How shall the requirements be represented in these work products (for example, natural-language-based or model-based, see below) and which notation(s) shall be used?

Requirements Engineers should define the RE work products to be used at an early stage in a project. Such early definition:

- Helps in the planning of efforts and resources
- Ensures that appropriate notations are used
- Ensures that all results are recorded in the right work products
- Ensures that no major reshuffling of information and "final editing" is needed
- Helps to avoid redundancy, resulting in less work and easier maintainability

3.2 Natural-Language-Based Work Products

Natural language, in both spoken and written form, has always been a core means for communicating requirements for systems. Using natural language to write RE work products has many advantages. In particular, natural language is extremely expressive and flexible, which means that almost any conceivable requirement in any aspect can be expressed in natural language. Furthermore, natural language is used in everyday life and is taught at school, so no specific training is required to read and understand requirements written in natural language.

Human evolution has shaped natural language as a means for *spoken communication between directly interacting people*, where misunderstandings and missing information can be detected and corrected rapidly. Hence, natural language is *not* optimized for precise, unambiguous, and comprehensive communication by means of written documents. This constitutes a major problem when writing technical documentation (such as requirements) in natural language. In contrast to communication in *spoken* natural language, where the communication is contextualized and interactive with immediate feedback, there is no natural means for rapidly detecting and correcting ambiguities, omissions, and

inconsistencies in texts *written* in natural language. On the contrary, finding such ambiguities, omissions, and inconsistencies in written texts is difficult and expensive, particularly for work products that contain a large amount of natural language text.

The problem can be mitigated to some extent by writing technical documentation consciously, following proven rules and avoiding known pitfalls.

When writing requirements in natural language, Requirements Engineers can avoid many potential misunderstandings by applying some simple rules:

- Write short and well-structured sentences. The rule of thumb is to express a single requirement in one sentence in natural language. To achieve a good structure, Requirements Engineers should use phrase templates (Section 3.33.3).
- Create well-structured work products. Besides writing well-structured sentences (see above), work products written in natural language should also be well-structured as a whole. A proven way to do this is by using a hierarchical structure of parts, chapters, sections, and subsections, as is usually done in technical books. Document templates (Section 3.3) help you to achieve a good structure.
- Define and consistently use a uniform terminology. Creating and using a glossary (Section 3.5) is the core means for avoiding misunderstandings and inconsistencies about terminology.
- Avoid using vague or ambiguous terms and phrases.
- Know and avoid the pitfalls of technical writing (see below).

When writing technical documents in natural language, there are some well-known pitfalls that should be avoided or things that need to be used with care (see, for example, [GoRu2003]).

Requirements Engineers should *avoid* writing requirements that contain the following:

- *Incomplete descriptions*. Verbs in natural language typically come with a set of placeholders for nouns or pronouns. For example, the verb "give" has three placeholders for *who* gives *what* to *whom*. When writing a requirement in natural language, all placeholders of the verb used should be filled.
- *Unspecific nouns*. Using nouns such as "the data" or "the user" leaves too much room for different interpretations by different stakeholders or developers. They should be replaced by more specific nouns or be made more specific by adding adjectives or assigning them a well-defined type.
- *Incomplete conditions*. When describing what shall be done, many people focus on the normal case, omitting exceptional cases. In technical writing, this is a trap to avoid: when something happens only if certain conditions are true, such conditions shall be stated, providing both *then* and *else* clauses.
- *Incomplete comparisons*. In spoken communication, people tend to use comparatives (for example, "the new video app is much better") without saying what they are comparing to, typically assuming that this is clear from the context. In technical writing, comparisons should include a reference object, for example, "faster *than 0.1 ms*".

There are some further things that Requirements Engineers need to use with care, as they constitute potential pitfalls:

- *Passive voice*. Sentences in passive voice have no acting subject. If a requirement is stated in the passive voice, this may hide who is responsible for the action described in the requirement, leading to an incomplete description.
- *Universal quantifiers*. Universal quantifiers are words such as *all*, *always*, or *never*, which are used to make statements that are universally true. In technical systems, however, such universal properties are rare. Whenever Requirements Engineers use a universal quantifier, they need to reflect on whether they are stating a truly universal property or whether they are instead specifying a general rule that has exceptions (which they also need to specify). They should apply the same caution when using "either–or" clauses, which, by their semantics, exclude any further exceptional cases.
- *Nominalizations*. When a noun is derived from a verb (for example, "authentication" from "to authenticate"), linguists call this a nominalization. When specifying requirements, Requirements Engineers need to handle nominalizations with care because a nominalization may hide unspecified requirements. For example, the requirement "Only after successful authentication, the system shall provide a user access to (...)" implies that a procedure for authenticating users exists. When writing such a requirement, therefore, the Requirements Engineer must check whether there are also requirements about the procedure for authenticating legitimate users.

Natural language is a very powerful means for writing requirements. To mitigate the inherent disadvantages of using natural language for technical documentation, Requirements Engineers should follow proven writing rules and avoid well-known pitfalls.

3.3 Template-Based Work Products

As mentioned in Section 3.2 above, using templates is a proven means for writing good, well-structured work products in natural language and thus mitigating some of the weaknesses of natural language for technical writing. A template is a kind of ready-made blueprint for the syntactic structure of a work product. When using natural language in RE, we distinguish between three classes of templates: phrase templates, form templates, and document templates.

3.3.1 Phrase Templates

> **Definition 3.2. Phrase template:**
> A template for the syntactic structure of a phrase that expresses an individual requirement or a user story in natural language.

A phrase template provides a skeleton structure with placeholders, in which Requirements Engineers fill in the placeholders in order to get well-structured, uniform sentences that express the requirements.

Using phrase templates is a best practice when writing individual requirements in natural language and when writing user stories.

3.3.1.1 Phrase Templates for Individual Requirements

Various phrase templates for writing individual requirements have been defined, for example, in [ISO29148], [MWHN2009], and [Rupp2014]. The standard ISO/IEC/IEEE 29148 [ISO29148] provides a single, uniform template for individual requirements as follows:

[<Condition>] <Subject> <Action> <Objects> [<Restriction>].

Example: When a valid card is sensed, the system shall display the "Enter your PIN" message on the dialog screen within 200 ms.

When formulating an action with this template, the following conventions about the use of auxiliary verbs are frequently used in practice:

- *Shall* denotes a mandatory requirement.
- *Should* denotes a requirement that is not mandatory but strongly desired.
- *May* denotes a suggestion.

Will (or using a *verb in the present tense* without one of the auxiliary verbs mentioned above) denotes a factual statement that is not considered as a requirement.

When there are no agreed meanings for auxiliary verbs in a project, or when in doubt, definitions such as the ones given above should be made part of a requirements specification.

EARS (Easy Approach to Requirements Syntax) [MWHN2009] provides a set of phrase templates that are adapted to different situations as described below.

Ubiquitous requirements (must always hold):

> The <system name> shall <system response>.

Event-driven requirements (triggered by an external event):

> WHEN <optional preconditions> <trigger> the <system name>
> shall <system response>.

Unwanted behavior (describing situations to be avoided):

> IF <optional preconditions> <trigger>, THEN the <system name>
> shall <system response>.

Note: Although the unwanted behavior template is similar to the event-driven one, Mavin et al. provide a separate template for the latter, arguing that unwanted behavior (primarily due to unexpected events in the context, such as failures, attacks, or things that nobody has thought of), is a major source of omissions in RE.

State-driven requirements (apply only in certain states):

> WHILE <in a specific state> the <system name> shall <system response>.

Optional features (applicable only if some feature is included in the system):

> WHERE <feature is included> the <system name> shall <system response>.

In practice, sentences that combine the keywords WHEN, WHILE, and WHERE may be needed to express complex requirements.

EARS has been designed primarily for the specification of cyber-physical systems. However, it can also be adapted for other types of systems.

3.3.1.2 Phrase Templates for User Stories

The classic phrase template for writing user stories was introduced by Cohn [Cohn2004]:

> As a <role> I want <requirement> so that <benefit>.

Example: "As a line manager, I want to make ad hoc inquiries to the accounting system so that I can do financial planning for my department."

While Cohn has designated the <benefit> part of the template as optional, it is standard practice nowadays to specify a benefit for every user story.

Every user story should be accompanied by a set of *acceptance criteria*—that is, criteria that the implementation of the user story must satisfy in order to be accepted by the stakeholders. Acceptance criteria make a user story more concrete and less ambiguous. This helps to avoid implementation errors due to misunderstandings.

3.3.2 Form Templates

Definition 3.3. Form template:
A template providing a form with predefined fields to be filled in.

Form templates are used to structure work products of medium size such as use cases. Cockburn [Cock2001] introduced a popular form template for use cases. [Laue2002] proposed a template for task descriptions.

Table 3.2 shows a simple form template for use cases. Each flow step may be subdivided into an action by an actor and the response by the system.

Table 3.2 A simple form template for writing use cases

Name	< A short active verb phrase>
Precondition	<Condition(s) that must hold when the execution of the use case is triggered>
Success end condition	<State upon successful completion of use case>
Failed end condition	<State upon failed execution of use case>
Primary actor	<Actor name>
Other actors	<List of other actors involved, if any>
Trigger	<Event that initiates the execution of the use case>
Normal flow	<Description of the main success scenario in a sequence of steps: <step 1> <action 1> <step 2> <action 2> ... <step n> <action n> ... >
Alternate flows	<Description of alternative or exceptional steps, with references to the corresponding steps in the normal flow>
Extensions	<Extensions to the normal flow (if there are any), with references to the extended steps in the normal flow>
Related information	<Optional field for further information, such as performance, frequency, relationship to other use cases, etc.>

Form templates are also useful for writing quality requirements in a measurable form [Gilb1988].

Table 3.3 provides a simple form template for measurable quality requirements, along with an example.

Table 3.3 A form template for specifying measurable quality requirements

Template		Example
ID	<Number of requirement>	R137.2
Goal	<Qualitatively stated goal>	Confirm room reservations immediately
Scale	<Scale for measuring the requirement>	Elapsed time in seconds (ratio scale)
Meter	<Procedure for measuring the requirement>	Timestamping the moments when the user hits the "Reserve" button and when the app has displayed the confirmation. Measuring the time difference.
Minimum	<Minimum acceptable quality to be achieved>	Less than 5 s in at least 95% of all cases
OK range	<Value range that is OK and is aimed at>	Between 0.5 and 3 s in more than 98% of all cases
Desired	<Quality achieved in the best possible case>	Less than 0.5 s in 100% of all cases

3.3.3 Document Templates

> **Definition 3.4. Document template:**
> A template providing a predefined skeleton structure for a document.

Document templates help to systematically structure requirements documents—for example, a system requirements specification. RE document templates may be found in standards, for example in [ISO29148]. The Volere template by Robertson and Robertson [RoRo2012], [Vole2020] is also popular in practice. When a requirements specification is included in the set of work products that a customer has ordered and will pay for, that customer may prescribe the use of document templates supplied by the customer. In Figure 3.1, we show an example of a simple document template for a system requirements specification.

3.3.4 Advantages and Disadvantages

Using templates when writing RE work products in natural language has major advantages. Templates provide a clear, re-usable structure for work products, make them look uniform, and thus improve the readability of the work products. Templates also help you to capture the most relevant information and make fewer errors of omission. On the other hand, there is a potential pitfall when Requirements Engineers use templates mechanically, focusing on the syntactic structure rather than on content, neglecting everything that does not fit the template.

Part	Sections
Part I: Introduction	
	System purpose
	Scope of system development
	Stakeholders
Part II: System overview	
	System vision and goals
	System context and boundary
	Overall system structure
	User characteristics
Part III: System requirements	
	Organized hierarchically according to system structure, using a hierarchical numbering scheme for requirements
	Per subsystem/component:
	Functional requirements (structure and data, function and flow, state and behavior)
	Quality requirements
	Constraints
	Interfaces
References	
	Glossary (if not managed as a work product of its own)
Appendices	
	Assumptions and dependencies

Figure 3.1 A simple system requirements specification template

Using templates when writing RE work products in natural language improves the quality of the work products provided that the templates are not misused as just a syntactic exercise.

3.4 Model-Based Work Products

Requirements formulated in natural language can easily be read by people provided they can speak the language. Natural language suffers from ambiguity due to the imprecision of semantics of words, phrases, and sentences [Davi1993]. This imprecision may lead to confusion and omissions in requirements. When you read textual requirements, you will try to interpret them in your own way. We often try to imagine these requirements in our mind. When the number of requirements is manageable, it is possible to maintain insight and an overview of the textual requirements. When the number of textual requirements becomes "too big," we lose the overview. That limit is different for each person. The number of textual requirements is not the only reason for losing insight and overview. The complexity of the requirements, the relationship between the requirements, and abstraction of the requirements also contribute to this. You may have to read the requirements formulated in natural language several times before you get a correct and complete picture that the system must comply with. We have a limited ability to process requirements in natural language.

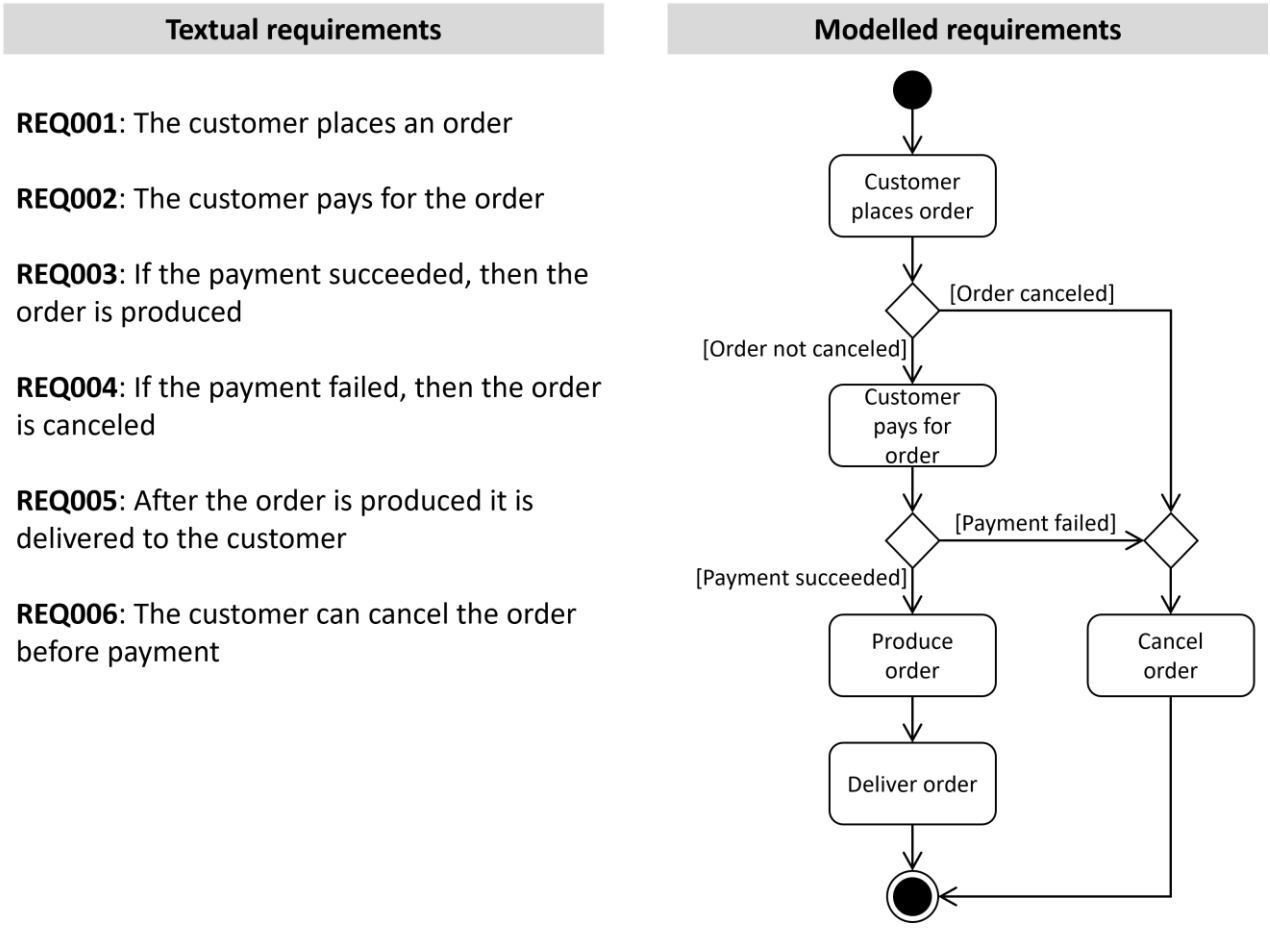


Figure 3.2 Textual requirements versus modeled requirements

A model is an abstract representation of an existing part of reality or a part of reality to be created. Displaying the requirements (also) with a model (or picture) will contribute to readers grasping the requirements. Such diagrammatic representation of a model is called a diagram.

The diagram in Figure 3.2 shows at a glance what the system must provide, but only if you have mastered the modeling language. It is evident that if you do not understand the diagram, in this case a UML activity diagram, the picture will not contribute to a better understanding of the requirements.

In the next section (3.4.1), the concept of a requirements model is explained. Modeling of business requirements and goals is explained in Section 3.4.6. An important method for describing the demarcation of a system is the context model. Examples of the context are depicted in Section 3.4.2. Sections 3.4.3 to 3.4.5 give a number of examples of modeling languages that are often used in systems engineering practice.

3.4.1 The Role of Models in Requirements Engineering

Like any language, a modeling language consists of grammatical rules and a description of the meaning of the language constructs, see Section 3.4.1.1. Although a model is a visual representation of reality, the language rules are important in order to understand the model and the nuances in the model.

It is not always efficient or effective to summarize the requirements in a model. By understanding the properties of a model, we can better determine when we can apply which model, see Section 3.4.1.2.

Just as natural language has advantages and disadvantages for expressing the requirements, so do models. If we observe these facts in applying a model, we can better determine the added value of applying the "correct" model. This is discussed in Section 3.4.1.3.

Many models have already been standardized and are used in various fields of application, see Section 3.4.1.4. Consider, for example, the construction of a house, where an architect uses a standardized model to describe the house. One example for models used by building architects are building information models (BIM) [ISO19650], that model the elements required to plan, build, and manage buildings and other construction elements.

Another example is electronics, where the drawing of electronic diagrams is standardized so that professionals can understand, calculate, and realize the electronics.

To determine whether a diagram is applied correctly, we can validate the quality criteria of a diagram. These criteria are described in Section 3.4.1.5.

3.4.1.1 Syntax and Semantics

If you think about a natural language, for example your native language, it is defined by its grammar and semantics.

The grammar describes the elements (words and sentences) and the rules that the language must obey. In a modeling language, this is called the syntax, see Figure 3.3. The syntax describes which notation elements (symbols) are used in the language. It also describes how these notation elements can be used in combination.

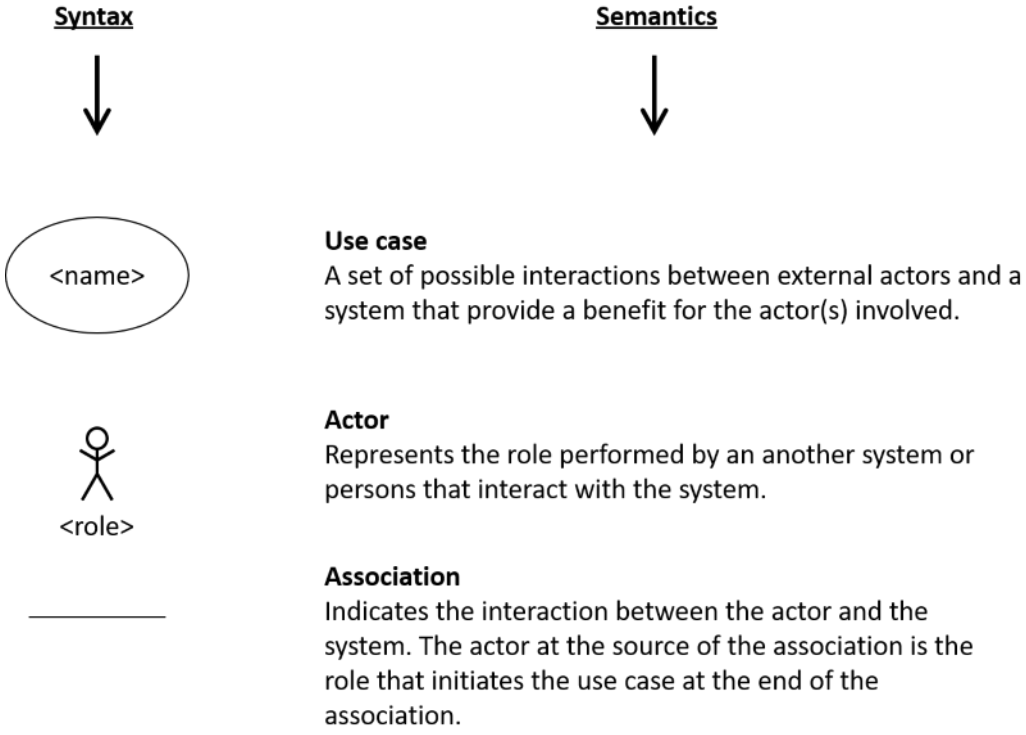


Figure 3.3 Modeling language syntax and semantics

The semantics defines the meaning of the notation elements and defines the meaning of the combination of elements. Understanding the meaning of the notation elements is fundamental for preventing the risk of the model being misinterpreted.

3.4.1.2 Properties of a Model

A requirements model is a conceptual model that depicts the requirements for the system to be developed. A model is also used to represent the current situation to understand, analyze, and explore the present problems. In this context, conceptual means that reality is reduced to its essence. A model has a high level of abstraction and reduces reality to what is relevant at this generic level.

A conceptual modeling language can be standardized (internationally) and is then referred to as a formal modeling language. An example of this is the widespread and frequently applied modeling language UML (Unified Modeling Language).

A model has a number of properties that are explored further in the following sections:

- A model is made for a specific purpose.
- A model gives a representation of reality.

- A model is used to reduce information so that we can better understand reality or focus on part of the reality.

A model is an abstract representation of an existing part of reality or a part of reality to be created. The notion of reality includes any conceivable set of elements, phenomena, or concepts, including other models. The modeled part of reality is called the original. The process to describe the original can be descriptive or prescriptive.

Modeling the existing original is called descriptive modeling. It shows the current reality and reflects the requirements that are met. If no model of the original is available yet, than such a model is the result of the analysis of the current situation.

Modeling an original to be created is called prescriptive modeling. It indicates what future reality is expected or required. If a model with descriptive properties exists for the given situation, then a model with prescriptive properties can be derived from the original by indicating which requirements will be new, changed, or are no longer needed. The prescriptive model describes the ultimate future situation desired.

Reality can be complex. If we apply "too many" details, a model can be hard to grasp. This complex reality can be simplified by reducing the amount of information in the model. In a model, we can omit irrelevant information. Reducing the amount of information can give us a better understanding of reality and allow us to understand the essence of this reality more easily. Based on the intended purpose (first property) for which the model is applied, only the relevant information is displayed in the model.

Please note, if "too much" information is reduced, a clouded or incorrect image of reality may arise. Thus, careful consideration should be given to how much of the information can be reduced without distorting reality.

There are several ways to reduce information:

- By compression or aggregation

  Aggregating information is a way to make information more abstract. The information is stripped of irrelevant details and is therefore more compact. The information is, as it were, condensed.

- By selection

  By selecting only the relevant information, and not everything, it is possible to indicate what the subject under consideration is. The focus is on a specific part or number of parts of the total.

Both ways of reducing information can also be applied together.

A model is a representation of reality and each model represents certain aspects of reality. For example, a construction drawing shows the breakdown of the space in a building and an electrical diagram shows the wiring of the electrical circuit.

Both models represent the building for a specific purpose. A model is made for a specific purpose in a specific context. In the example above, the context is the design and/or realization of a building. The various construction drawings represent information about a

specific aspect of the building. This makes it immediately clear that a specific model can be used only if it fits the purpose for which the model was made.

3.4.1.3 Advantages and Disadvantages of Modeling Requirements

Compared with natural languages, models have the following advantages, among others:

- The elements and their connections are easier to understand and to remember.

  A picture tells more than a thousand words. A picture, and also a model, can be easier to grasp and to remember. Note that a model is not self-explanatory and needs extra information—i.e., a legend, examples, scenarios, etc.

- The focus on a single aspect reduces the cognitive load needed to understand the requirements modeled.

  Because a model has a specific purpose and a reduced amount of information, understanding the reality modeled can require less effort.

- Requirements modeling languages have a restricted syntax that reduces possible ambiguities and omissions.

  Because the modeling language (syntax and semantics) is simpler—i.e., limited number of notation elements and stricter language rules compared with natural language—the risk of confusion and omissions is smaller.

- Higher potential for automated analysis and processing of requirements.

  Because a modeling language is more formal (limited number of notation elements and stricter language rules) than a natural language, it lends itself better to automating the analysis or processing of requirements.

Despite the great advantages for visualizing requirements with models, models also have their limitations.

- Keeping models that focus on different aspects consistent with each other is challenging.

  If multiple models are used to describe the requirements, it is important to keep these models consistent with each other. This requires a lot of discipline and coordination between the models.

- Information from different models needs to be integrated for causal understanding.

  If multiple models are used, all models must be understood to enable a good understanding of the requirements.

- Models focus primarily on functional requirements.

  The models for describing quality requirements and constraints are limited if not lacking in specific context. These types of requirements should then be supplied in natural language together with the models—for example, as a separate work product.

- The restricted syntax of a graphic modeling language implies that not every relevant item of information can be expressed in a model.

  Because a model is made for a specific purpose and context, it is not always possible to record all requirements in the model or in multiple models. Requirements that cannot be expressed in models are added to the model as natural language requirements or as a separate work product.

Therefore, requirements models should always be accompanied by natural language [Davi1995].

3.4.1.4 Application of Requirements Models

As indicated in the previous sections, there are common models for various contexts. For example, in architecture, you have construction drawings, piping diagrams, electrical diagrams, etc. to express the specifications of a building. In other contexts—for example, software development—there are modeling languages that are useful in these types of context. An important aspect in applying models is to use models that are common in the context or that have been specially developed for a specific context.

Many modeling languages—for example, UML [OMG2017] or BPMN [OMG2013]—have been standardized. When requirements are specified in a non-standard modeling language, the syntax and semantics of the language should be explained to the reader—for example, via a legend.

Models are used to describe the requirements from a certain perspective. In system development, functional requirements are categorized in the following perspectives (see also Section 3.1.4):

- Structure and data

  Models that focus on the static structural properties of a system or a domain

- Function and flow

  Models that focus on the sequence of actions required to produce the required results from given inputs or the actions required to execute a (business) process, including the flow of control and data between the actions and who is responsible for which action

- State and behavior

  Models that focus on the behavior of a system or the life cycle of business objects in terms of state-dependent reactions to events or the dynamics of component interaction

The nature of the system being modified or built gives direction to the models to be used. For example, if the nature of the system is to process information and relationships, then it is expected that there are quite a lot of functional requirements that describe this information and these relationships. As a result, we use a matching modeling language that lends itself to modeling data and its structure.

Naturally, a system will consist of a combination of the above perspectives. It follows that a system needs to be modeled from multiple perspectives. Sections 3.4.3 to 3.4.5 elaborate the different models for each perspective in more detail.

Before the requirements are elicited and documented—for example with models—an inventory is taken of goals and context. These can also be modeled, see Sections 3.4.6 respectively 3.4.2.

Applying models helps us mainly in the following ways:

- *Specifying* (primarily functional) requirements in part or even completely, as a means of replacing textually represented requirements
- *Decomposing* a complex reality into well-defined and complementing aspects; each aspect being represented by a specific model, helping us to grasp the complexity of the reality
- *Paraphrasing* textually represented requirements in order to improve their comprehensibility, in particular with respect to relationships between them
- *Validating* textually represented requirements with the goal of uncovering omissions, ambiguities, and inconsistencies

Modeling the requirements also helps with structuring and analyzing knowledge. You can use diagrams to structure your own thoughts to get a better understanding of the system and its context.

3.4.1.5 Quality Aspects of a Requirements Model

This is a supplementary section for which there will be no questions in the CPRE Foundation level exam.

A substantial part of the requirements models are diagrams or graphical representations. The quality of the requirements model is determined by the quality of the individual diagrams and their mutual relationships. In turn, the quality of the individual diagrams is determined by the quality of the model elements within the diagrams.

The quality of the requirements models and model elements can be assessed against three criteria [LiSS1994]:

- Syntactic quality
- Semantic quality
- Pragmatic quality

The syntactic quality expresses the extent to which a single model element (graphical or textual), requirements diagram, or requirements model complies with the syntactic specifications. If, for example, a model that describes the requirements as a class model contains modeling elements that are not part of the syntax, or model elements are misused, then this will decrease the syntactic quality of the model. A stakeholder of this model—for example, a tester—might misinterpret the information that is represented by the model. This might eventually lead to inappropriate test cases.

Requirements modeling tools provide facilities for checking the syntactic quality of the models.

The semantic quality expresses the extent to which a single model element (graphical or textual), the requirements diagram, or the requirements model correctly and completely represents the facts.

Just like in natural language, semantics gives meaning to the words. If a term can have different meanings or there are several terms that mean the same thing, this can lead to miscommunication. The same applies to the semantics of modeling elements. If the modeling elements are misinterpreted or applied incorrectly, the model may be misinterpreted.

The pragmatic quality expresses the extent to which a single model element (graphical or textual), the requirements diagram, or the requirements model is suitable for the intended use—that is, whether the degree of detail and abstraction level is appropriate for the intended use and whether the appropriate model is selected with respect to the domain or context. This can be assessed if the purpose and the stakeholders of the diagram are known. Intermediate versions of the model can be submitted to the stakeholders interested to validate whether the diagrams fit their purpose.

During validation of the requirements, the quality of the modeling diagrams used is assessed to make sure that these diagrams fit their intended purpose and usefulness.

3.4.1.6 Best of Both Worlds

As explained in the previous section, requirements that are expressed in textual or visual/graphical form (i.e., via requirements models) have their advantages and disadvantages. By using both textual and graphic representations of the requirements, we can harness the power and benefits of both forms of representation.

Amending a model with textual requirements adds more meaning to the model. Another useful combination is that we can link quality requirements and constraints to a model or specific modeling element. This provides a more complete picture of the specific requirements.

Using models can also support the textual requirements. Adding models and images to the textual requirements supports these models for a better understanding and overview.

3.4.2 Modeling System Context

Chapter 2, Principle 4 introduces the notion that requirements never come in isolation and that the system context, such as existing systems, processes, and users need to be considered when defining the requirements for the new or changed system.

Context models specify the structural embedding of the system in its environment, with its interactions to the users of the system as well as to other new or existing systems within the relevant context. A context model is not a graphical description of the requirements but is used to reveal some of the sources of the requirements. Figure 3.4 provides an abstract

example of a system and its environment, with its interfaces to the users of the system and its interfaces to other systems. Thus, context diagrams help to identify user interfaces as well as system interfaces. If the system interacts with users, the user interfaces must be specified in a later step during RE.

If the system interacts with other systems, the interfaces to these systems must be defined in more detail in a later step. Interfaces to other systems may already exist or may need to be developed or modified.

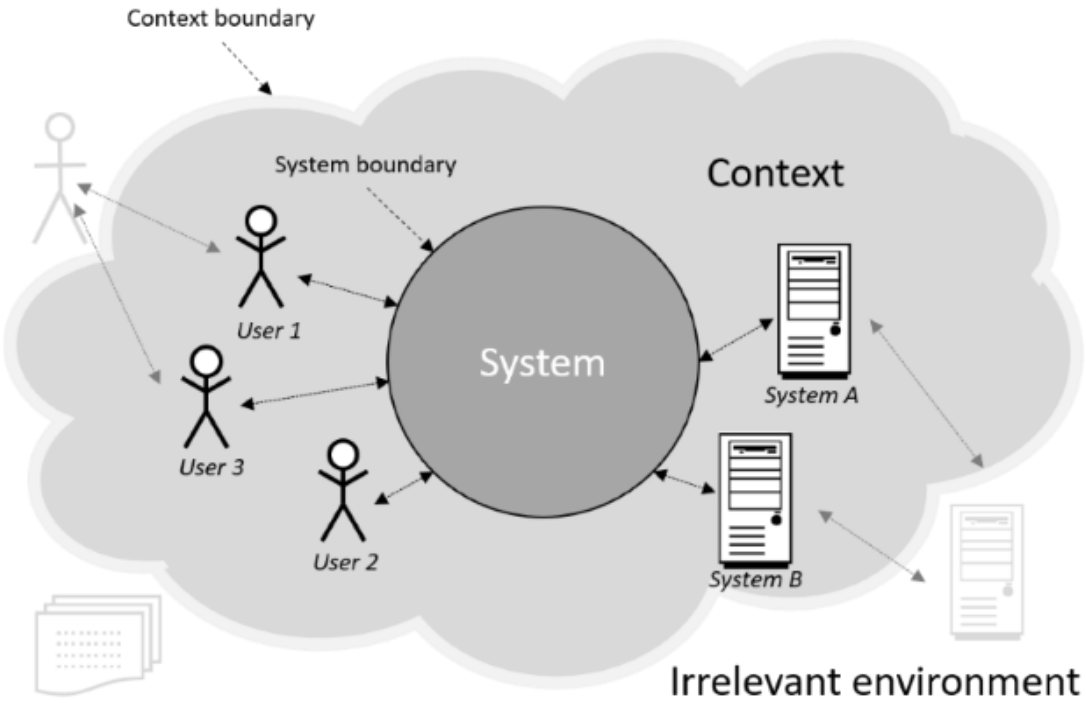


Figure 3.4 A system in its context

Even if there is no standardized modeling language for context models, context models are frequently represented by:

- Data flow diagrams from structured analysis [DeMa1978]
- UML use case diagrams [OMG2017]
- Note: the UML use case model consists of two elements; the UML use case diagram (see Figure 3.6) and the use case specification (Section 3.4.2.2). This chapter focuses on modeling with the UML use case diagrams.
- Tailored box-and-line diagrams [Glin2019]

In the systems engineering domain, SysML block definition diagrams [OMG2018] can be adapted to express context models by using stereotyped blocks for the system and the actors.

In the next two subsections, we introduce the notation of data flow diagrams (DFD) and UML use case diagrams to model the context of a system. These two examples do not describe the complete context but emphasize the context from a specific viewpoint.

3.4.2.1 Data Flow Diagram

The system context can be viewed from different perspectives. The structured analysis of systems [DeMa1978] talks about the context diagram. This diagram is a special data flow diagram (DFD) where the system is represented by one process (the system). Figure 3.5 shows an example of a context diagram.

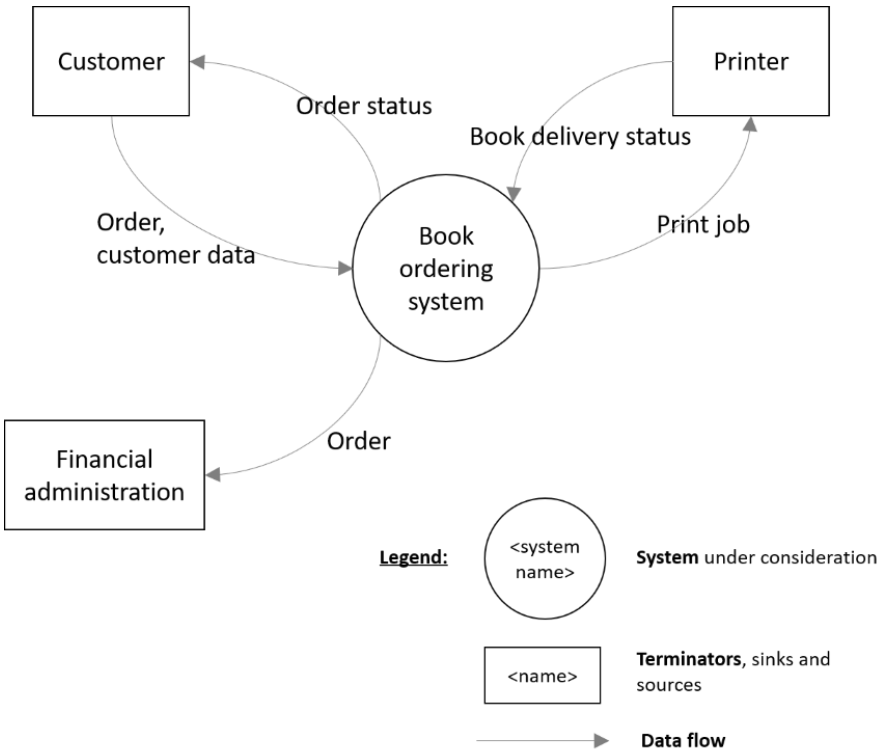


Figure 3.5 Example of a context diagram using a DFD

The system is placed centrally in the model. It has a clear name so that the readers know which system is being considered.

The rectangles around the system are terminators: customer, printer, and financial administration. A terminator that provides information or services to the system is called a *source*. A terminator that takes information or services from the system is called a *sink*. A terminator can take either role depending on the data provided or retrieved, such as the customer in the example above.

The arrows in the example show how the information from the terminators flows into the system (source) and from the system to the terminators (sinks). The arrows are given a logical name that describes what information is transferred. Irrelevant details are omitted at the context diagram level. The information flow between the customer and the system contains, for example, *customer data*. What information (name, date of birth, email address,

telephone number, delivery address, billing address, etc.) makes up the *customer data* does not have to be relevant yet for this level of abstraction.

The flow of information can consist of tangible (materials) and intangible (information) objects. Also, at this conceptual level, there is no reference (yet) to *how*—email, website, form, etc.—the information is provided.

Adding extra details to the context diagram can make it clearer to the stakeholders involved and may help to improve the shared understanding. These details need to be worked out for each individual situation.

Using a data flow diagram to model the context of a system provides some insights into the interactions of the system with its environment, for example:

- The interfaces to people, departments, organizations, and other systems in the environment
- The (tangible and intangible) objects that the system receives from the environment
- The (tangible and intangible) objects that is produced by the system and is delivered to the environment

A data flow diagram indicates a clear boundary between the system and its environment. The relevant users and systems of the environment are identified during elicitation of requirements (Section 4.1). DFD context diagrams can help to structure the context to reach a shared understanding of the system context and the system boundary.

3.4.2.2 UML Use Case Diagram

Another view of the context of a system can be reached from a functional perspective. The UML use case diagram is a common approach for modeling the functional aspects of a system and the system boundaries, along with the system's interactions with users and other systems. Use cases provide an easy way to systematically describe the various functions within the defined scope from a user perspective. This is different to DFD context diagrams, where the system is represented as a big black box.

Use cases were first proposed as a method for documenting the functions of a system in [Jaco1992]. The UML use cases consists of use case diagrams with associated textual use case specifications (see Section 3.3.2). A use case specification specifies each use case in detail by, for example, describing the possible activities of the use case, its processing logic, and preconditions and postconditions of the execution of the use case. The specification of use cases is essentially textual—for example, via use case templates as recommended in [Cock2001].

As mentioned, a UML use case diagram shows the functions (use cases) from the point of view of the direct users and other systems that interact with the system under consideration. The name of the use case is often composed of a verb and a noun. This gives a brief description of the function offered by the system, as shown by the example in Figure 3.6.

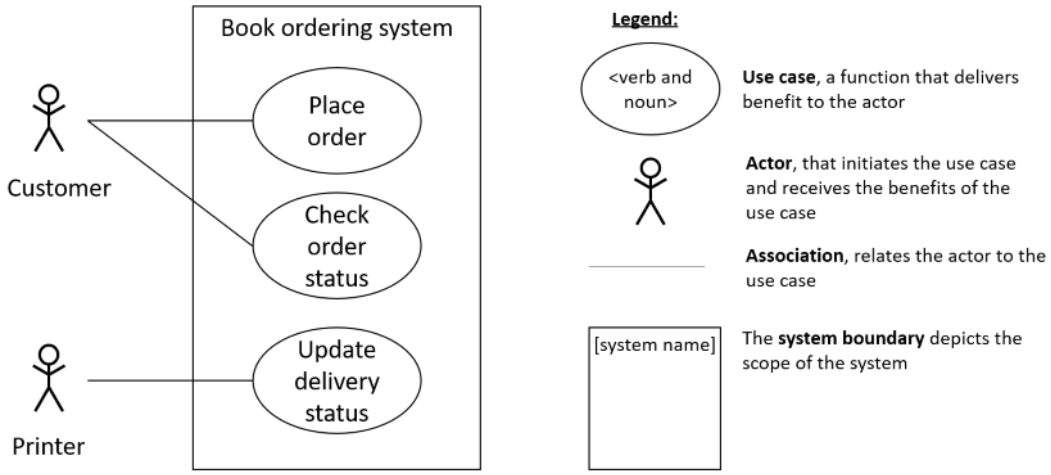


Figure 3.6 Example of a context diagram using a UML use case diagram

The actors are the direct users or systems that interact with the system under consideration. The actor (user or system) that starts the use case receives the benefit that the use case delivers (e.g., showing the status of an order to the customer). The association connects the actor with the relevant use case but it does not document any direction or data flow (as is done in DFDs); it expresses only that the actor receives the benefit from the use case.

A UML use case diagram describes the functionality that the system offers to its environment. The separation between the functionality in the system and the actors in the context is visualized with the system boundary (rectangle around the use cases, e.g., "book ordering system"). Use case diagrams support sharpening of the system boundary and checking whether the functional scope of the system at a high level is covered.

Each use case also includes a detailed use case specification, documenting the preconditions, trigger, actions, postconditions, actors, and so forth. Use cases are usually described using a template (Section 3.3). If the scenarios of a use case become complex or large, the recommendation is to visualize the scenarios with UML activity diagrams, see Section 3.4.4.1. The detailed specification of use cases is not part of context modeling and can be elaborated at a later time, when this information becomes relevant.

3.4.3 Modeling Structure and Data

For functional requirements from the perspective of business objects (see Section 3.1.4), different data models are available. A (business) object can be a tangible or intangible object, such as a bicycle, pedal, bicycle bell, but also a training request, a shopping basket with digital products, and so on. A (business) object is "something" in the real world. Some (or maybe all) of these (business) objects are used by the system under consideration. The system uses these objects as input to process, to persist, and/or to deliver output. Data models are used to describe the (business) objects that must be known by the system. These kinds of diagrams model the object, attributes of the object, and the relationships between objects. For the sake of simplicity, we refer to modeling structure and data—these, however, represents information structures between (business) objects in the real world.

A number of common models for depicting structure and data are:

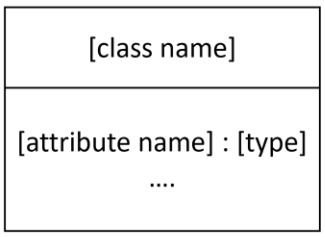
- Entity relationship diagrams (ERD) [Chen1976]
- UML class diagrams [OMG2017]. See Section 3.4.3.1
- SysML Block Definition Diagrams [OMG2018]. See Section 3.4.6.2

To explain the concept of modeling structure and data, this chapter uses the UML class diagram as an example. UML, short for Unified Modeling Language, consists of an integrated set of diagrams. This set of diagrams is a collection of best engineering practices and has proven successful in modeling complex and large systems. UML was designed by Grady Booch, James Rumbaugh, and Ivar Jacobson in the 1990s and it has been a standardized modeling language since 1997.If more depth or a different model is desired, read the literature referred to and practice with the desired modeling language.

3.4.3.1 UML Class Diagrams

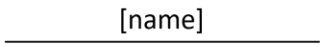
UML is a collection of different models that can be used to describe a system. One of these models is the class diagram. A class diagram depicts a set of classes and associations between them. We discuss only the common and simple notation elements of this model. If more depth is desired, we refer to the literature or the CPRE Advanced Level Requirements Modeling.

In the overview below you will find the most common notation elements.

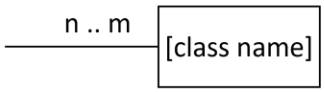


A **class** describes a set of (material or immaterial) objects that have a similar structure, behaviour and relationships.

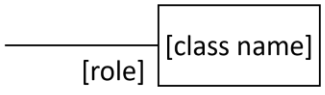
An **attribute** describes a property of the class. The attribute is expressed as a certain **type** that restricts the values that can be assigned to the attribute.



An **association** relates two classes to each other.



Multiplicities define how many instances of the class at the corresponding association end can participate in the association to a given instance of the class at the other end of the association. Where $n \in \mathbb{N}$ and $m \in \mathbb{N}$. Some examples are:
0 .. 1 (zero or one time)
0 .. * (zero or more times)
1 .. * (one or more times)
7 (exactly seven times)
1 (default value, exactly one time)



The **role** name defines which role an object of the class plays in the association

Figure 3.7 Subset of the modeling elements of UML class diagrams

In a class model, you will find the concepts and terms that are relevant in the domain. These concepts include a clear definition that is included in the glossary. With the use of data models, the glossary is extended with information about the structure and coherence of the terms and concepts. A clear definition and coherence of the terms used prevents miscommunication about the matter under consideration.

Figure 3.8 shows a simplified model of the book ordering system (see examples of the context in Section 3.4.2). The static information that the system needs to perform its functionality—ordering a book—is modeled.

A customer orders a book and hence information is persisted for the classes *Customer*, *Order,* and *Book*. A customer can place an order and therefore a relationship (association) exists between the *Customer* and the *Order*. A customer can place multiple orders over time and he/she only becomes a customer if he/she places an order. This information determines the multiplicity: 1 customer places 1 or more orders.

The fact that a customer can order a book means that there is also a relationship between the classes *Order* and *Book*. To keep the example simple, here, the customer can order only one book at a time. Also, an order must contain at least one book. An order that has no book is not an order.

In the class *Book,* the attribute *inStock* is also maintained. Information such as "if the stock is not sufficient to fulfill the order, then a print job is sent to the printer" cannot be modeled. This is a type of information that cannot be modeled in a class diagram because it describes a certain functionality of the system. This information is part of the requirements and should be documented in another work product. It can be added as a textual requirement that accompanies the class diagram, or be modeled with another diagram—for example, a UML activity diagram (see Section 3.4.4.1).

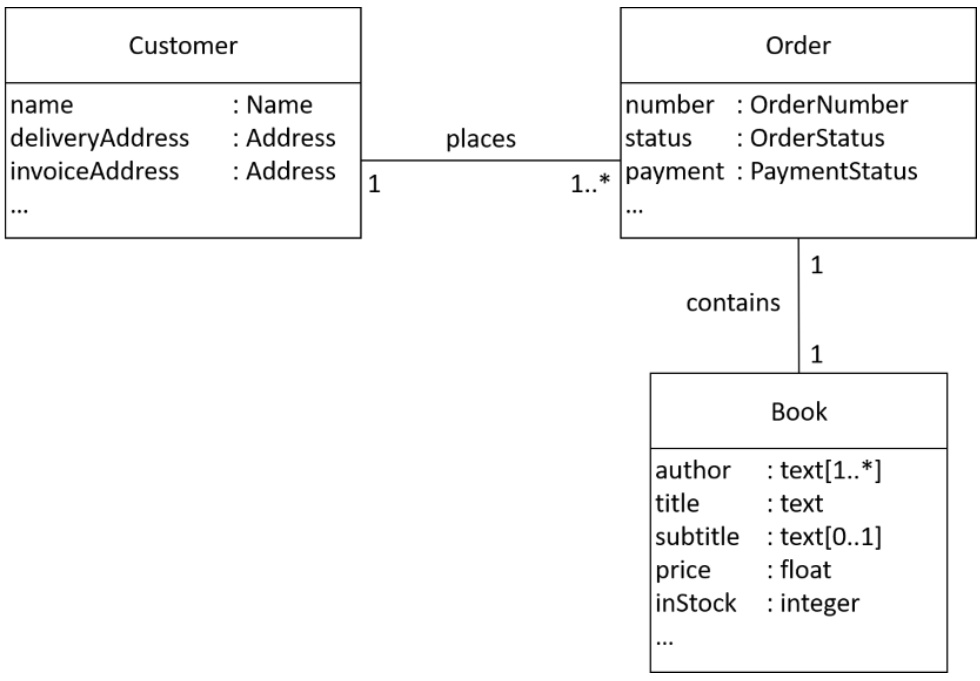


Figure 3.8 Example of a simple UML class diagram

3.4.4 Modeling Function and Flow

Function and flow describe how the (sub)system shall transform input into output. We can visualize this type of requirement with models that depict function and flow.

Unlike modeling data, which essentially needs only one diagram type, function and flow can be viewed from different angles. Depending on the needs of the stakeholders to take the next step in the development process, more than one model might be needed to document the requirements about function and flow.

Some common models for depicting function and flow are:

- UML use case diagram [OMG2017]. See Section 3.4.2.2
- UML activity diagram [OMG2017]. See Section 3.4.4.1
- Data flow diagram [DeMa1978]. See Section 3.4.2.1
- Domain story models [HoSch2020]. See Section 3.4.6.3
- Business Process Modeling Notation (BPMN) [OMG2013].

Excurse: BPMN process models are used to describe business processes or technical processes. BPMN is frequently used to express business process models.

To explain the concept of modeling function and flow, we limit this section to a few examples of UML diagrams. If more depth or a different model is desired, read the literature referred to and practice with the relevant modeling language.

3.4.4.1 UML Activity Diagram

UML activity models are used to specify system functions. They provide elements for modeling actions and the control flow between actions. Activity diagrams can also express who is responsible for which action. Advanced modeling elements (not covered by this handbook) provide the means for modeling data flow.

A UML activity diagram expresses the control flow of activities of a (sub)system. Flow thinking comes from visualizing program code with flow charts (according to [DIN66001], [ISO5807]). This helped programmers to conceive and understand complex structures and flows in programs. With the introduction of UML [OMG2017], a model has been introduced for visualizing activities and actions from a functional perspective.

In the overview below you will find the basic notation elements.

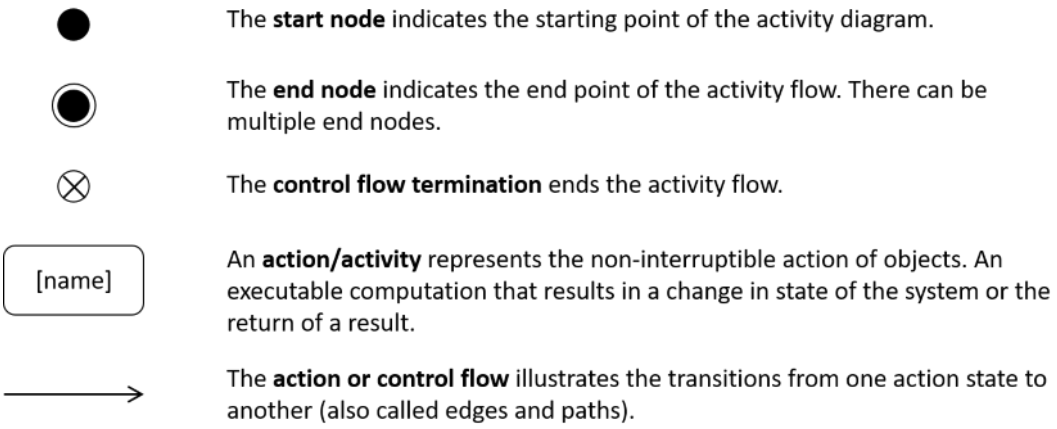


Figure 3.9 Basic notation elements of the UML activity diagram

With this set of basic notation elements, you can set up a simple sequential activity diagram. If more control is required, the model can be extended with decisions and parallel flows of activities using the notation elements below.

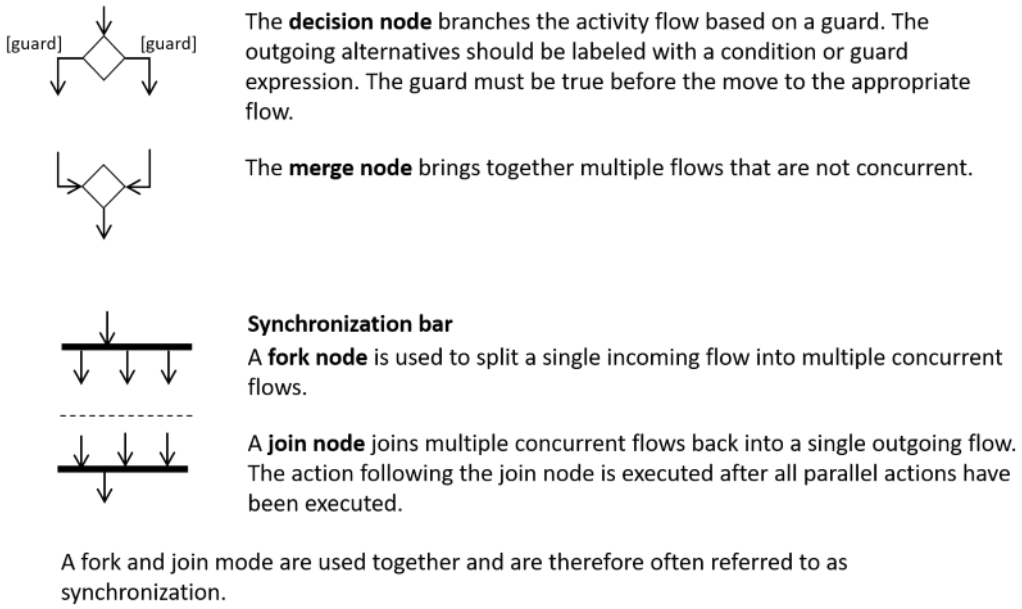


Figure 3.10 Decisions and parallel flows in a UML activity diagram

Activity diagrams can be used to specify the processing logic of use case scenarios in detail (see Section 3.3.2). Activity diagrams are created to visualize the scenarios, which are processes with activities and processing logic. As long as the diagram remains understandable, the main scenario can be modeled jointly with the alternative scenarios and the exception scenarios as part of the same diagram.

Figure 3.11 gives a simple example of the book ordering system. This simplified flow of action starts when the customer sends in their order. First, the *Order* and the *Customer* information are validated to determine whether all (necessary) information is supplied. If either the *Order* or the *Customer* information is invalid (incorrect or insufficient), then a notification is sent to

the customer and the order process is canceled. The basic scenario is that the *Order* and *Customer* information are valid. The scenario that the *Order* or *Customer* information is invalid is called an exceptional flow and handles a functional faulty condition in the process.

If both *Order* and *Customer* information are correct, then the stock is checked. If there is a sufficient number of products in stock, the *Order* is picked and sent to the *Customer*. An alternative flow is started if there are insufficient products in stock. A print job request is sent to the *Printer* and a notification for a redelivery is sent to the *Customer*.

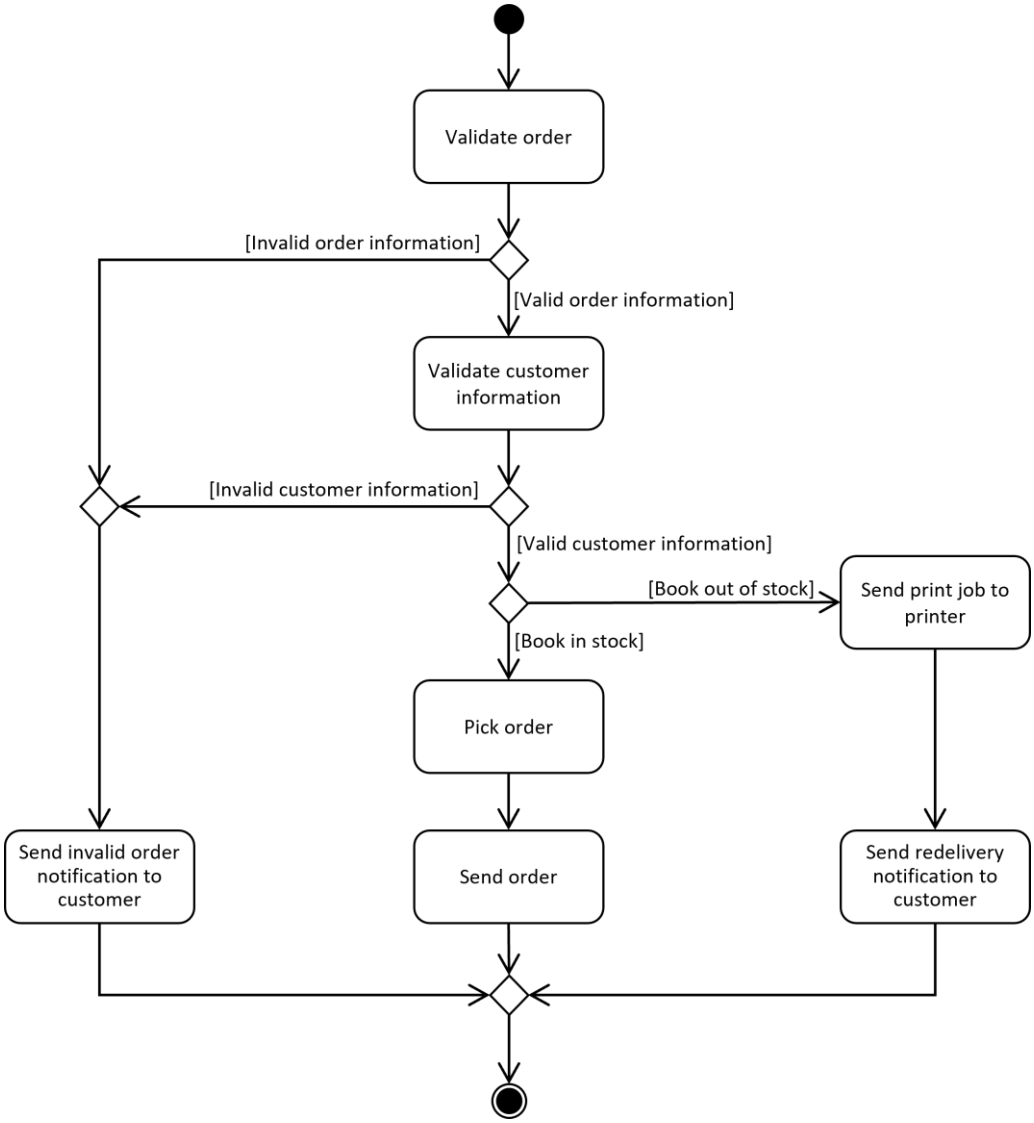


Figure 3.11 Example of a UML activity diagram

Within the book ordering system, there are also other flows that are separated from the order and delivery process. For example, the payment, redelivery, and invoice processes have separate flows to allow a clear separation of concerns. If, for example, the decision is taken to no longer keep any products in stock, then the order and delivery process still applies. If changes are needed in this flow, these changes may not affect the other flows. This decomposition of functionality helps to keep things simple and clear.

3.4.5 Modeling State and Behavior

Functional requirements that describe the behavior, states, and transitions of a (sub)system or that of a business object are requirements in the behavioral perspective. An example of a system state is *On*, *Standby*, or *Off*. A business object can have a life cycle that goes through a number of prescribed states. For example, a business object *Order* can be in the following states: *Placed*, *Validated*, *Paid*, *Shipped*, and *Completed*.

A technique widely used to describe the behavior of a system is statecharts [Hare1988]. Statecharts are state machines with states that are decomposed hierarchically and/or orthogonally. State machines, including statecharts, can be expressed in the UML modeling language [OMG2017] with state machine diagrams (also called state diagrams).

State diagrams describe state machines that are finite. This means that these systems eventually reach a final state. A state diagram shows the states that the system or an object can take. It also indicates how to switch state—that is, the state transition. A system does little by itself. Switching the state requires a trigger from the system or from the environment of the system.

Common models for representing behavior and states include:

- Statecharts [Hare1988]
- UML state diagram [OMG2017]

3.4.5.1 UML State diagram

To explain the concept of modeling behavior and states, this chapter uses the UML state diagram as an example. If more depth or a different model is desired, read the literature referred to and practice with the relevant modeling language.

In the overview below you will find the basic notation elements.

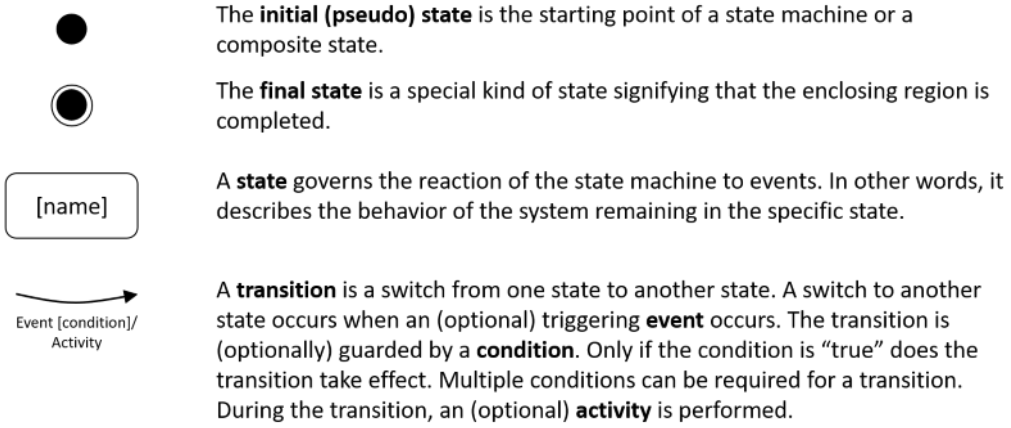


Figure 3.12 Basic notation elements of the UML state diagram

As discussed at the beginning of the section, a state diagram can clarify the states an object can take. We see here an opportunity to visualize additional (and partly redundant)

information of an object. Imagine that you order a book on a website and you want to track the status of your order. An order is used in the real world and is modeled as a business object in a class diagram (see Figure 3.8) with, most likely, an attribute *status*. The class diagram indicates that the attribute *status* can assume a limited number of values, such as *Validated*, *Paid*, *Delivered*, *Canceled*, and so on. The class diagram does not describe the order of possible status changes. A class diagram does not describe the behavior of the system in a certain "status" either. This can be made clear with a UML state diagram—for example, that an offered order cannot go directly to the status *Delivered* without the customer having paid for the order.

Figure 3.13 gives an example of a state diagram of the book ordering system. In the class diagram (Figure 3.8) of the book ordering system, an object *Order* is modeled. This object has an attribute *status* that can have a limited number of values. These values are listed and explained in the class diagram. What a class diagram does not describe is the sequence in which the order is processed. A state diagram visualizes the states and transitions between the states, making it clear what the sequence of the order status is. The state diagram shows, for example, that the order cannot be sent before it is completely picked (transition between the states *Picked* and *Sent*). Also, if the order is in the state *Sent,* the next state can only be *Paid*. A transition from *Sent* to *Handled* is not possible. This diagram also makes clear that payment happens after the book is sent. You can ask the stakeholders whether this is what they need or have requested.

A transition may direct to the same status. This situation is visible in the state *Picked*. Each time the order is not picked to completion, it stays in the same state to prevent it from sending an incomplete order. Only when the order is completely picked is it then sent to the customer.

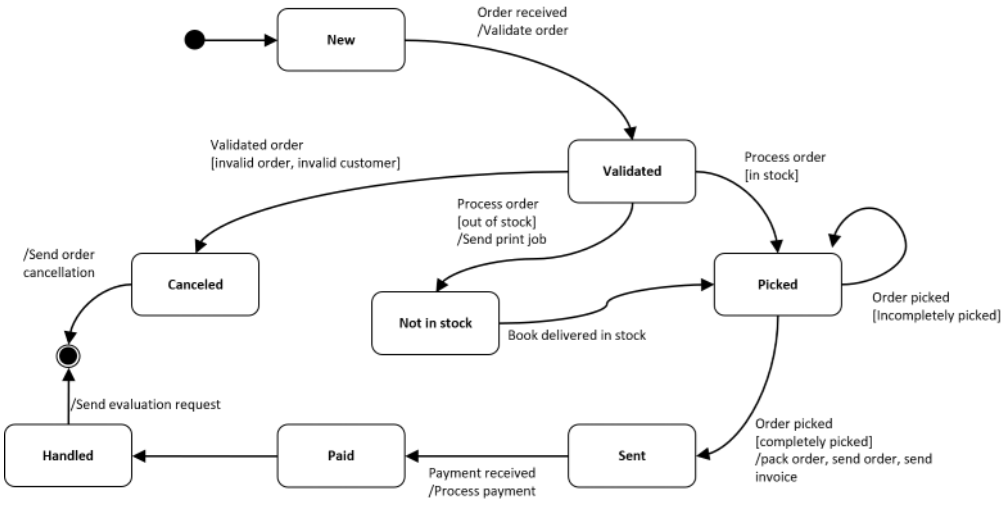


Figure 3.13 Example of a UML state diagram

A few months after the release of the book ordering system, customers complained that they did not have the ability to cancel an order. It was agreed that a customer could cancel the order in each state of the order process. Modeling this new requirement means that a

transition to *Canceled* is needed from each state. This might make the diagram difficult to read. Adding a textual requirement to describe this behavior might be a way to keep the model simple for the audience.

3.4.6 Supplementary models

At the CPRE foundation level, the understanding and application of models is restricted to selected, important model types. There are further model types that are used in Requirements Engineering. The following subsections provide some additional examples for models that are supplementary and will not be questioned in the CPRE Foundation level exam.

3.4.6.1 Modeling Goals

Business requirements describe a business goal or need. They describe the end result that the solution must meet and with which the (business) problem is solved, see Chapter 2, Principle 5. To ensure that the focus is on solving the problem and that the effort focuses on adding value, goals are carefully described. In Requirements Engineering, there are several ways to document goals. The most common one is the use of natural language (Section 3.2) or templates (Section 3.3). Template-based documentation forms can be found, for instance, in [Pich2010], [Pohl2010], or [RoRo2012].

There are also some model-based notations for documenting goals. The easiest and most common notation is an AND/OR tree [AnPC1994]. AND/OR trees allow us to document goals at different levels of detail and to link subgoals with goals using AND and OR relationships. An AND relationship means that all subgoals need to be fulfilled to fulfill the goal. An OR relationship is used to express that at least one of the subgoals needs to be fulfilled to fulfill the goal.

More elaborate modeling approaches for goals can be found in:

- Goal-oriented requirements language (GRL) [GRL2020]

  This is a language that supports goal-oriented modeling and reasoning of requirements, especially for dealing with non-functional requirements.

- Knowledge acquisition in automated specification (KAOS) [vLam2009]

  KAOS is a methodology that contains goal modeling. This enables analysts to build requirements models and to derive requirements documents from KAOS goal models.

- The i* framework is one of the most popular goal- and agent oriented modelling and reasoning methods in the field. i* supports the creation of models representing an organization or a socio-technical system. [FLCC2016] provides a comprehensive overview of the i* framework and it application.

Documenting goals (in textual or graphical form) is an important starting point for eliciting requirements, referring the requirements to their rationale, and identifying sources—like stakeholders—of the requirements, etc.

3.4.6.2 SysML block definition diagrams

Systems Modeling Language (SysML) [OMG2018] is a general-purpose modeling language for systems engineering applications. SysML is a dialect of UML, which re-uses and extends parts of UML.

SysML can be adopted for many different purposes. *Block definition diagrams* in SysML are an extension of the UML class diagram. They can, for example, be adapted to express context diagrams by using stereotyped blocks for the system and the actors. Block definition diagrams can also be used to model the structure of a system in terms of the system's conceptual entities and the relationships between them.

3.4.6.3 Domain story models

Domain story models can be used to model function and flow, by specifying visual stories about how actors interact with devices, artifacts and other items in a domain, typically using domain-specific symbols [HoSch2020]. They are a means for understanding the application domain in which a system will operate.

The techniques is meant to be very simply executable for telling a story, a board and sticky notes could be enough. Gathering the relevant stakeholders who really know how the business operates provoke meaningful discussion by telling stories that occur in the domain. Domain story telling improves the shared understanding of a business process, and is used to analyze and solve problems in the domain.

3.4.6.4 UML Sequence Diagram

The UML sequence diagram is used to depict the interaction between communication partners and to model the dynamic aspect of systems. The dynamic aspect of systems that a sequence diagram depict can be function & flow as well a state & behavior. Therefore, a UML sequence diagram can be used for different purposes.

The communication partners in a UML sequence diagram are actors, systems, components, and/or objects within a system. The interaction displays the sequence of messages (a scenario) between these communication partners. The interaction that takes place between the communication partners realizes the purpose of a scenario, respectively (a part) of a use case.

In the overview below you will find the basic notation elements.

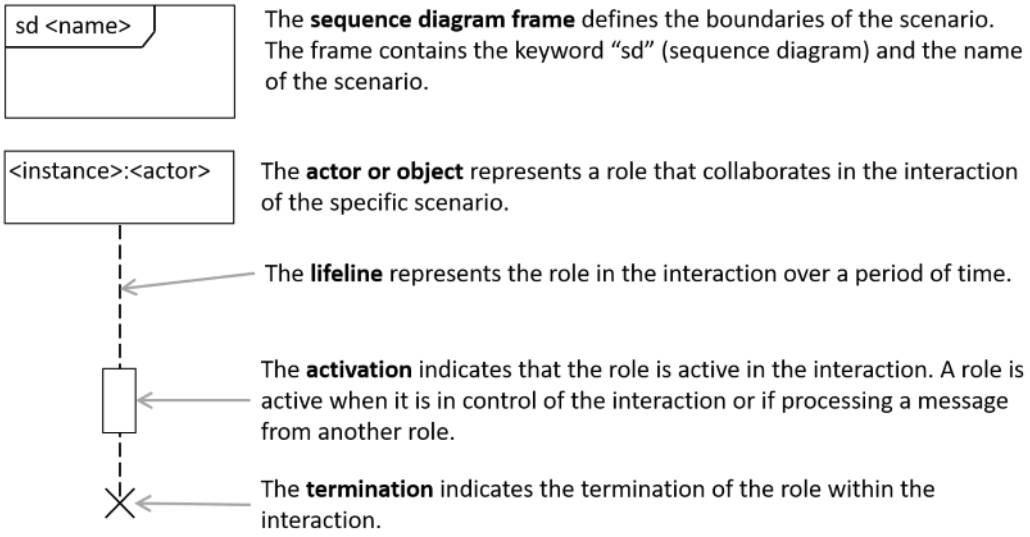


Figure 3.14 Basic notation elements of the UML sequence diagram

A lifeline in a scenario depicts the role in the scenario, meaning the instance of an actor. When sequence diagrams are modeled, the instance name of an actor or object is often omitted. The roles that participate in the communication interact with each other by sending messages. There are two types of messages that are used in the interaction.

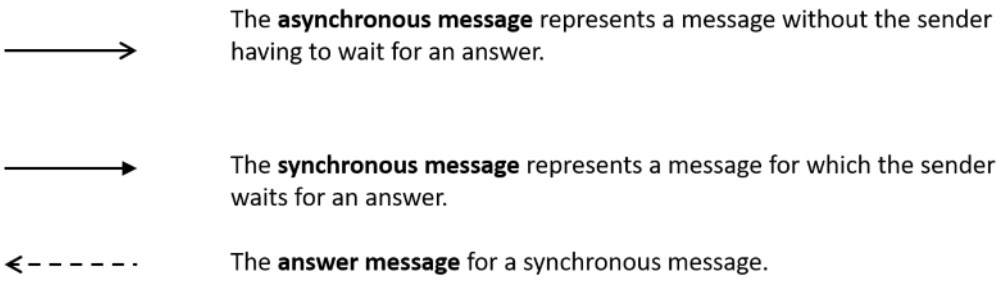


Figure 3.15 Basic notation elements of messaging in the UML sequence diagram

A message can also be sent from or to objects outside the scenario. This is represented as a filled-in circle. The sender or receiver of these kinds of messages may be unknown.

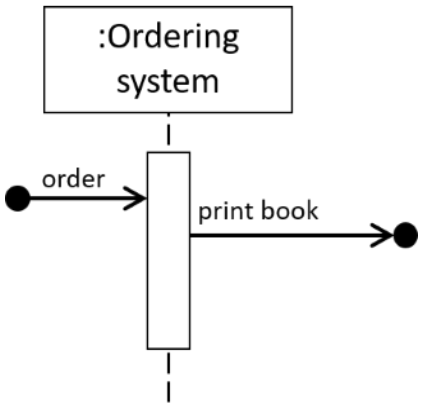


Figure 3.16 Messages from and to an object outside the scenario

Figure 3.17 shows a model of the scenario in which a *customer* orders a book and that specific book is out of stock. The *Customer* asks to place an *Order*. If the *Order* is invalid, a notification that the *Order* is canceled is returned. If the *Order* is valid, the stock is checked and if a book is out of stock, a print job is sent to the *Printer*.

This is a synchronous message because we are awaiting to receive the book – even it might take some time to print the book. A notification is sent to the *Customer* that the book is out of stock and will be redelivered. The *Order* is deactivated until the book is delivered by the *Printer*.

When the book is received from the *Printer,* the *Ordering system* is activated again. The order is picked and sent to the *Customer*. This completes the *Order* and a last notification of the status is sent to the *Customer*.

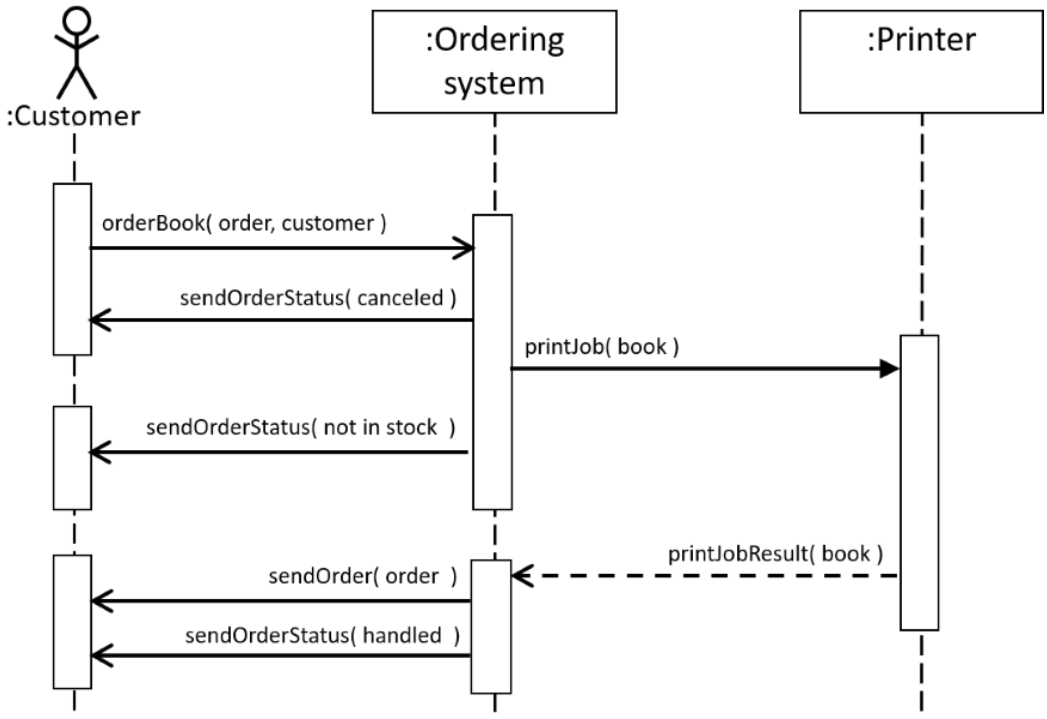


Figure 3.17 Example of a UML sequence diagram

3.5 Glossaries

Glossaries are a core means of establishing shared understanding of the terminology used when developing a system: they help avoid people involved as stakeholders or developers using and interpreting the same terms in different ways.

A good glossary contains definitions for all terms that are relevant for the system, be they context-specific terms or everyday terms that are used with a special meaning in the context of the system to be developed. A glossary should also define all abbreviations and acronyms used. If there are synonyms (that is, different terms denoting the same thing), they should be marked as such. Homonyms (that is, identical terms that denote different things) should be avoided or at least marked as such in the glossary.

There are a couple of rules that guide the creation, use, and maintenance of the glossary in a system development project.

- *Creation and maintenance*. To ensure that the terminology defined in the glossary is consistent and always up to date, it is vital that the glossary is managed and maintained centrally over the entire course of a project, with one person or a small group being responsible for the glossary. When defining terms, it is important that the stakeholders are involved and agree on the terminology.
- *Usage*. In order to get the full benefit of a glossary, its use must be mandatory. Work products should be checked for proper glossary usage. Obviously, this means that everybody involved in a project must have read access to the glossary.

When an organization develops related systems in multiple projects, it makes sense to create a glossary at the enterprise level in order to achieve consistent terminology across projects.

Creating, maintaining, and using a glossary consistently avoids errors and misunderstandings concerning the terminology used. Working with glossaries is a standard best practice in RE.

3.6 Requirements Documents and Documentation Structures

It is not sufficient to work with requirements at the level of individual requirements. Requirements must be collated and grouped in suitable work products, be they explicit requirements documents or other RE-related documentation structures (such as a product backlog).

Document templates (see Section 3.3.3) may be used to organize such documents with a well-defined structure in order to create a consistent and maintainable collection of requirements. Document templates are available in literature [Vole2020], [RoRo2012] and in standards [ISO29148]. Templates may also be reused from previous, similar projects or may be imposed by a customer. An organization may also decide to create a document template as an internal standard.

A requirements document may also contain additional information and explanations—for example, a glossary, acceptance criteria, project information, or characteristics of the technical implementation.

Frequently used requirements documents are:

- *Stakeholder requirements specification*: the stakeholders' desires and needs that shall be satisfied by building a system, seen from the stakeholders' perspective. When a customer writes a stakeholder requirements specification, it is called a *customer requirements specification*.
- *User requirements specification*: a subset of a stakeholder requirements specification, covering only requirements of stakeholders who are prospective users of a system.
- *System requirements specification*: the requirements for a system to be built and its context so that it satisfies its stakeholders' desires and needs.
- *Business requirements specification*: the business goals, objectives, and needs of an organization that shall be achieved by employing a system (or a collection of systems).
- *Vision document*: a conceptual imagination of a future system, describing its key characteristics and how it will create value for its users.

Frequently used alternative documentation structures are:

- *Product backlog*: a prioritized list of work items, covering all requirements that are needed and known for the product
- *Sprint backlog*: a selected subset of a product backlog with work items that will be realized in the next iteration
- *Story map*: a visual two-dimensional organization of user stories in a product backlog with respect to time and content

There is no standard or universal requirements document or documentation structure. Accordingly, documents or documentation structures should not be reused from previous projects without reflection.

- The actual choice depends on several factors, for example:
- The development process chosen
- The project type and domain (for example, tailor-made solution, product development, or standard product customizing)
- The contract (a customer may prescribe the use of a given documentation structure)
- The size of the document (the larger the document, the more structure is needed)

3.7 Prototypes in Requirements Engineering

Prototypes play an important role both in engineering and design.

> **Definition 3.5 Prototype:**
> 1. In manufacturing: A piece which is built prior to the start of mass production.
> 2. In software and systems engineering: A preliminary, partial realization of certain characteristics of a system.
> 3. In design: A preliminary, partial instance of a design solution.

Prototypes in software and systems engineering are used for three major purposes [LiSZ1994]:

Exploratory prototypes are used to create shared understanding, clarify requirements, or validate requirements at different levels of fidelity. Such prototypes constitute temporary work products that are discarded after use. Exploratory prototypes may also be used as a means of specification by example. Such prototypes must be treated as evolving or durable work products.

Experimental prototypes (also called breadboards) are used to explore technical design solution concepts, in particular with respect to their technical feasibility. They are discarded after use. Experimental prototypes are not used in RE.

Evolutionary prototypes are pilot systems that form the core of a system to be developed. The final system evolves by incrementally extending and improving the pilot system in several iterations. Agile system development frequently employs an evolutionary prototyping approach.

Requirements Engineers primarily use exploratory prototypes as a means for requirements elicitation and validation. In elicitation, prototypes serve as a means of specification by example. In particular, when stakeholders cannot express what they want clearly, a prototype can demonstrate what they would get, which helps them shape their requirements. In validation, prototypes are a powerful means for validating the *adequacy* (see Section 3.8) of requirements.

Exploratory prototypes can be built and used with different degrees of fidelity. We distinguish between wireframes, mock-ups, and native prototypes.

Wireframes (also called paper prototypes) are low-fidelity prototypes built with paper or other simple materials that serve primarily for discussing and validating design ideas and user interface concepts. When prototyping digital systems, wireframes may also be built with digital sketching tools or dedicated wireframing tools. However, when using a digital tool for wireframing, it is important to retain the essential properties of a wireframe: it can be built quickly, modified easily, and does not look polished nor resemble a final product.

Mock-ups are medium-fidelity prototypes. When specifying digital systems, they use real screens and click flows but without real functionality. They serve primarily for specifying and validating user interfaces. Mock-ups give users a realistic experience of how to interact with a system through its user interface. They are typically built with dedicated prototyping tools.

Native prototypes are high-fidelity prototypes that implement critical parts of a system to an extent that stakeholders can use the prototype to see whether the prototyped part of the system will work and behave as expected.

They serve both for specification by example and for thorough validation of critical requirements. Native prototypes may also be used to explore and decide about requirements *variants* for some aspect—for example, two different possible ways of supporting a given business process.

Depending on the degree of fidelity, exploratory prototypes can be an expensive work product. Requirements Engineers have to consider the trade-off between the cost of building and using prototypes and the value gained in terms of easier elicitation and reduced risk of inadequate or even wrong requirements.

3.8 Quality Criteria for Work Products and Requirements

Obviously, Requirements Engineers should strive to write good requirements that meet given quality criteria. RE literature and standards provide a rich set of such quality criteria. However, there is no general consensus about which quality criteria shall be applied for requirements. The set of criteria presented in this subsection aims to provide a proven practice at foundation level.

Modern RE follows a value-oriented approach to requirements (see Principle 1 in Chapter 2). Consequently, the degree to which a requirement fulfills the given quality criteria shall correspond to the value created by this requirement. This has two important consequences:

- Requirements do not have to fully adhere to all quality criteria.
- Some quality criteria are more important than others.

We distinguish between quality criteria for single requirements and quality criteria for RE work products such as RE documents or documentation structures.

For single requirements, we recommend using the following quality criteria:

- *Adequate*: the requirement describes true and agreed stakeholder needs.
- *Necessary*: the requirement is part of the relevant system scope, meaning that it will contribute to the achievement of at least one stakeholder goal or need.
- *Unambiguous*: there is a true shared understanding of the requirement, meaning that everybody involved interprets it in the same way.
- *Complete*: the requirement is self-contained, meaning that no parts necessary for understanding it are missing.
- *Understandable*: the requirement is comprehensible to the target audience, meaning that the target audience can fully understand the requirement.

- *Verifiable*: the fulfillment of the requirement by an implemented system can be checked indisputably (so that stakeholders or customers can decide whether or not a requirement is fulfilled by the implemented system).

Adequacy and understandability are the most important quality criteria. Without them, a requirement is useless or even detrimental, regardless of the fulfillment of all other criteria. Verifiability is important when the system implemented must undergo a formal acceptance procedure.

Some people use *correctness* instead of *adequacy*. However, the notion of correctness implies that there is a formal procedure for deciding whether something is correct or not. As there is no formal procedure for validating a documented requirement against the desires and needs that stakeholders have in mind, we prefer the term adequacy over correctness.

For work products covering multiple requirements, we recommend applying the following quality criteria:

- *Consistent*: no two requirements, recorded in a single work product or in different work products, contradict each other.
- *Non-redundant*: each requirement is documented only once and does not overlap with another requirement.
- *Complete*: the work product contains all relevant requirements (functional requirements, quality requirements, and constraints) that are known at this point in time and that are related to this work product.
- *Modifiable*: the work product is set up in such a way that it can be modified without degrading its quality.
- *Traceable*: the requirements in the work product can be traced back to their origins, forward to their implementation (in design, code, and test), and to other requirements they depend on.
- *Conformant*: if there are mandatory structuring or formatting instructions, the work product must conform to them.

3.9 Further Reading

Mavin et al. [MWHN2009] introduce and describe the EARS template. Robertson and Robertson [RoRo2012] describe the Volere templates. Goetz and Rupp [GoRu2003], [Rupp2014] discuss rules and pitfalls for writing requirements in natural language. Cockburn [Cock2001] has written an entire book about how to write use cases. Lauesen [Laue2002] discusses task descriptions and also provides some examples of real-world RE work products.

The ISO/IEC/IEEE standard 29148 [ISO29148] provides many resources concerning RE work products: phrase templates, quality criteria for requirements, and detailed descriptions of the content of various RE work products, including a document template for every work product. Cohn [Cohn2010] has a chapter on how to frame requirements in a product backlog.

Gregory [Greg2016] and Glinz [Glin2016] discuss the problem of how detailed requirements should be specified and to what extent complete and unambiguous requirements specifications are possible.

Numerous publications deal with using models to specify requirements. The UML specification [OMG2017], as well as textbooks about UML, describe the models available in UML. Hofer and Schwentner [HoSch2020] introduce domain modeling with domain storytelling. [OMG2013] and [OMG2018] describe the modeling languages BPMN for modeling business processes and SysML for modeling systems, respectively. The books by Booch, Rumbaugh, and Jacobson [BoRJ2005], [JaSB2011], [RuJB2004] give more depth and (practical) applications of UML. Furthermore, the following books and articles are recommended for more thorough knowledge and patterns in modeling requirements: [DaTW2012], [Davi1993], [Fowl1996], [GHJV1994]. [LiSS1994] and [Pohl2010] provide a better understanding of the quality aspects of models.

4 Practices for Requirements Elaboration

In the previous chapters, we learned about the nature of requirements as the representation of the wishes and needs of people and organizations for something new (e.g., a system to be developed or adapted), about the principles that underlie the production of the requirements, and about ways to capture the requirements in documentation. We establish requirements before we build or modify a (part of a) system to ensure that the system is useful for—and accepted by—the people or the organization that requested it. These requirements then serve as input for a development team that will build and implement the system.

This is Requirements Engineering in a nutshell; it happens, explicitly or often implicitly, whenever and wherever people try to develop something. In principle, the quality of the requirements determines the quality of the output of the subsequent development. Without proper requirements, it is unlikely that the resulting system will be useful. Therefore, it is important to elaborate the requirements in a professional way. This necessitates an explicit definition of the *how to*: the practices to be used for high-quality elaboration.

That is what this chapter is about: it gives an overview of the tasks, activities, and practices that are relevant for anyone involved in Requirements Engineering. It starts with the search for potential sources of requirements and it ends with the delivery of a single, consistent, understandable, and agreed set of requirements that can serve as input for the efficient development, maintenance, and operation of an effective system.

The first task in every Requirements Engineering effort will be identifying and analyzing potential *sources for requirements*. This may seem like a simple and obvious task, but as we will see in Section 4.1, there are quite a few aspects that need to be considered and analyzed. Overlooking a source will inevitably lead to poor or even missing requirements and therefore degrade the quality of the resulting system.

The next step is eliciting the requirements from these sources. It is like drawing water from a well: you never know what is in the bucket until you have brought it to the surface. In Requirements Engineering, this task is called *elicitation*; it is explained in Section 0. In elicitation, we turn implicit desires, wishes, needs, demands, expectations, and whatever else into explicit requirements that can be recognized and understood by all parties involved.

However, when you ask two people about their requirements for a certain system, you will rarely get exactly the same answers. In a whole series of requirements elicited from different sources, it is almost certain that some of them will be conflicting. As it is impossible to implement conflicting requirements in one and the same system, *conflict resolution* will always be an important task in Requirements Engineering, as described in Section 4.3.

Section 4.4 is devoted to the final task in Requirements Engineering: the *validation of requirements*. The purpose of this step is to ensure that the quality of the set of requirements elicited and the individual requirements within this set is good enough to enable subsequent system development.

From the above description of Requirements Engineering tasks, you could get the impression that they are performed as a linear process with a strict sequence of steps. However, this is certainly not the intention of this description and rarely the case in practice.

Figure 4.1 shows some process steps that are common in Requirements Engineering. They might be performed in parallel, in loops, or sequentially—whatever is suitable in the given situation.

The starting point is often a limited set of obvious sources. During elicitation from these sources, new sources are identified, triggering additional elicitation tasks. When conflicts are encountered, more detailed elicitation may be required to find a way out. In validation, it may appear that a source has been overlooked, a requirement is faulty, or a conflict has remained uncovered, resulting in a new series of source analysis, elicitation, and/or conflict resolution and validation activities. Even during the subsequent system development, circumstances may necessitate additional Requirements Engineering.

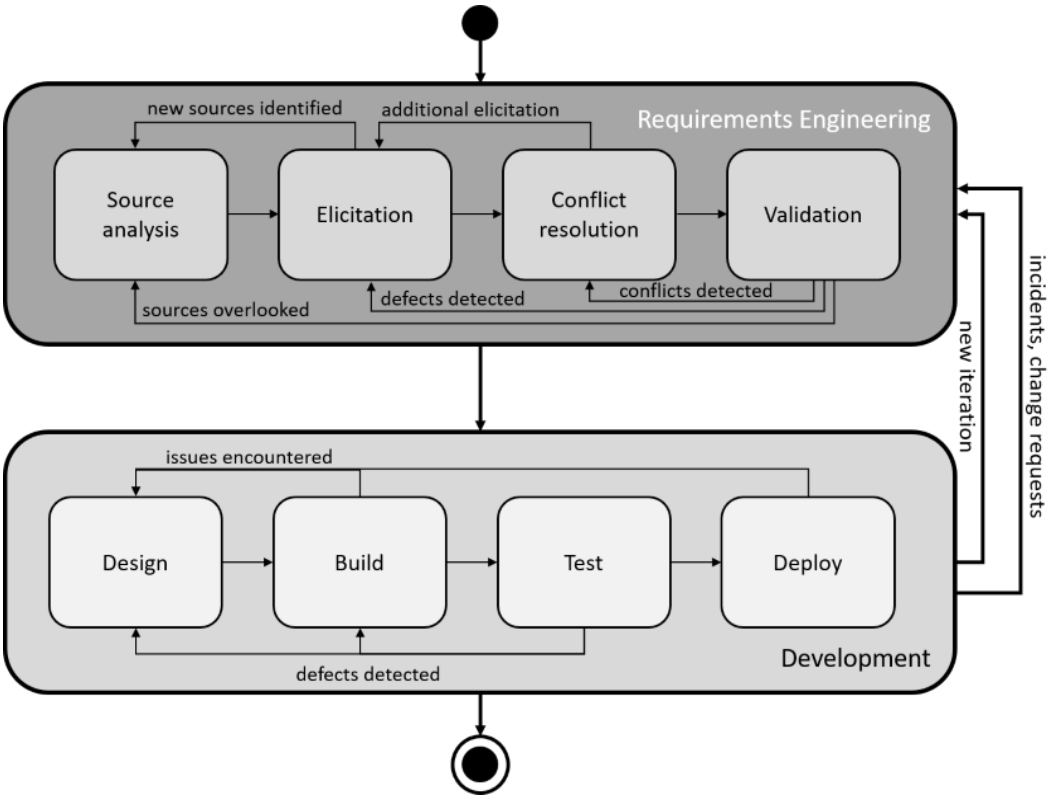


Figure 4.1 Requirements Engineering is not a linear process

In agile projects, iterative and incremental Requirements Engineering and system development go hand in hand, with requirements being elaborated just before the development of a new system increment. In such projects, you will often see that a project starts with a limited product backlog of high-level requirements that are refined and detailed only when they are candidates for the next iteration.

4.1 Sources for Requirements

Requirements are not like candy bars, lying on the shelf for everyone to pick them as they please. In the introduction to this chapter, we compared requirements with water to be drawn from a well: it is quite an effort to bring them to the surface. Therefore, the first problem that a Requirements Engineer will face is "Where are the wells?" As no requirement comes without a source, one of the first activities in requirements elicitation is to identify the potential sources. It is not enough to identify these sources only at the beginning of a project or product development; this is a process that will be repeated over and over again.

Right from the start of requirements elaboration, the Requirements Engineer should be engaged in identifying, analyzing, and involving all relevant requirements sources, as missing a relevant source will inevitably lead to an incomplete understanding of relevant requirements. And this will continue until the end: the identification of requirements sources is a process that requires constant reconsideration.

Chapter 2, Principle 3 emphasizes the necessity for (explicit and implicit) *shared understanding* between and among all parties involved: stakeholders, Requirements Engineers, developers. Understanding the context of the system to be developed in a certain application domain is a prerequisite to being able to identify the relevant requirements sources. Domain knowledge, previous successful collaboration, common culture and values, and mutual trust are enablers for shared understanding, while geographic distance, outsourcing, or large teams with high turnover are obstacles.

In Chapter 2, Principle 4, we introduced the context as a concept that is essential for understanding and specifying a system and its requirements. We defined the context as that part of reality that lies between the *system boundary* and the *context boundary*. Entities in this context will somehow influence the system or even interact with it but will not be contained in the system itself.

This would make the search for requirements sources quite simple: just look around in the context! But it is not that easy. At the start of a development process, the context has not been defined yet; the system boundary and the context boundary still have to be determined. Therefore, the search for requirements sources is an iterative, recursive process.

Potential sources are analyzed for their relationship with the future system. If you find no relationship when analyzing a potential source, this means that it is part of the irrelevant environment and will not be analyzed for requirements. Potential sources that appear to be part of the future system are of no interest to the Requirements Engineer either; they *belong* to the developers. Only those entities for which analysis reveals an interaction with, an interface to, or an influence on the future system, but that remain (relatively) unchanged during the next development deserve attention as sources for requirements. In this analysis, the system boundary and the context boundary are delineated, vague at first and becoming sharper as more and more sources are identified. As the context thus becomes clearer, it becomes easier to identify new sources, which in turn sharpen the boundaries further.

The search for requirements sources usually starts with a few obvious sources, often identified by the client at the start of a development effort. Initial elicitation from these sources will uncover other potential sources, which are then analyzed to decide whether or not they are relevant for the system. During this analysis, new potential sources may again pop up. In fact, in every elicitation effort, the Requirements Engineer will remain keen on detecting new sources. This may continue until the very end of the development effort. However, we try to identify the major, most relevant sources early, because all other Requirements Engineering activities depend on this early identification.

In Requirements Engineering, we discern three major categories of sources:

- Stakeholders
- Documents
- (Other) systems

These categories are discussed in more detail in the following sections.

4.1.1 Stakeholders

In Chapter 2, Principle 2, you learned about the stakeholder as a person or organization that *influences* a system's requirements or *is impacted* by that system.

The stakeholders of a system are the main sources for requirements. Even more than with other sources, failure to include a relevant stakeholder will have a major negative impact on the quality of the final set of requirements; discovering such stakeholders late (or missing them altogether) may lead to expensive changes or, at the end, a useless system. To create a system that fulfills the needs of all of its stakeholders, the systematic identification of stakeholders should start at the beginning of any development effort and the results should be managed throughout development. Stakeholders can be found in a broad area around the system, ranging from direct and indirect users of the system, (business) managers, IT staff such as developers and operators, to opponents and competitors, governmental and regulatory institutions, and many others. The prime question for identifying a person or an organization as a stakeholder is: "Does a relevant relationship exist between the person/organization and the system?"

> It helps to see stakeholders as human beings made of flesh and blood. If you identify an organization as stakeholder, ask yourself questions such as the following: "Can I identify a person who is responsible for this organization? Who can be seen as the prime contact of this organization? Who represents this organization within our company?" For instance, if the government is the stakeholder because a certain law is involved, look for someone who represents the government as the source to be approached for requirements. In this case, it is not very useful to identify the Prime Minister as this person; the head of the internal legal department would be a better choice.

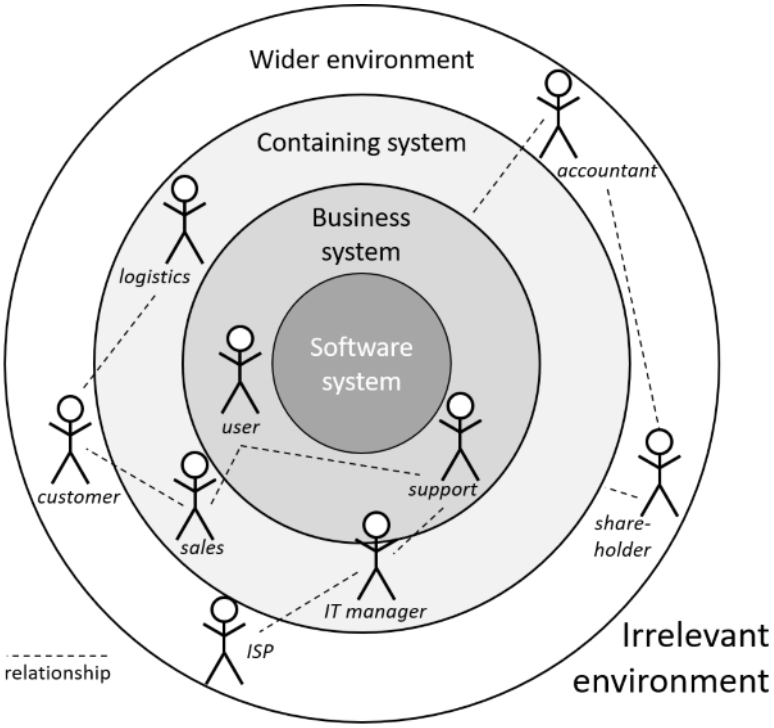


Figure 4.2 Alexander's onion model

There is no standard technique for identifying stakeholders but Ian Alexander's onion model [Alex2005] can be a good start, see Figure 4.2. This model shows how a (software) system is surrounded by several layers of higher-level socio-technical[1] and social systems, each having its own stakeholders. At the start of a requirements development effort, a few of these stakeholders will be evident—for instance, end users or customers. They can be used as a starting point in the search for other stakeholders. After identifying them as relevant sources, the Requirements Engineer will analyze their relationships, both in inner and outer surrounding systems. In this analysis, new stakeholders will be found, who in turn may have other (and more) relationships to be analyzed. You could call this the *snowball* principle: the more stakeholders you have found, the easier it becomes to find new ones. However, when arriving at stakeholders in Alexander's *wider environment*, any outer relationships will end up in the irrelevant environment, which means that they will no longer reveal new sources.

Apart from stakeholders referring to other stakeholders, documents can often reveal new stakeholders. Good examples are organizational charts, process descriptions, marketing reports, and regulatory documents. For more information about documentation as a source for requirements, see Section 4.1.2. Checklists of typical stakeholder groups and roles can be a useful tool to avoid overlooking certain inconspicuous potential stakeholders. Also, analyzing stakeholders of legacy or similar systems can help.

[1] A socio-technical system is a system that considers requirements spanning hardware, software, personal, and community aspects, while recognizing the interaction between society's complex infrastructures and human behavior.

As a Requirements Engineer, you will collect a lot of data about your stakeholders and maintain this data until your work is done. You must know who the stakeholders are, how you can reach them, when and where they are available, what their expertise is, as well as their relevance as a source, what their attitude towards the project is and their influence on it, what their roles are in the company, in the project and in their relation to the system, etc.

Usually, this information is kept in a stakeholders list, and it must be kept up to date, as during all steps, you will remain in contact with all stakeholders—some intensely and very closely, others infrequently and superficially. See Table 4.1 below for a simplified example.

Table 4.1 Example of a stakeholders list

Name	Dept	Phone	Avail-ability	Role	Influence	Interest
Marlene	Owner	482263	Mondays only	Sponsor	++	o
Peter	Sales	481225	Permanent	Product owner	++	+
Eva	Legal	481237	Not in June	Consultant	+	–
Hassan	Logistics	242651	Permanent	User	o	++
Mira	Service desk	242424	After 4pm	User	–	+
Natalia)*		481226	Permanent	Customer	++	++

* Persona, created, maintained and represented by *customer panel* team

Maintaining a good, open relationship with the stakeholders is key to getting relevant information from them. This relies primarily on behavioral characteristics such as integrity, honesty, and respect.

Open and frequent communication about your plans, your activities, and your results is essential. You may have to turn stakeholders from initial opponents into collaborators. As a Requirements Engineer, you must understand what the stakeholders expect from you. You must also *sell* your work by showing them the benefits of your solution and by removing the impediments that stakeholders experience or expect on their way to that solution. Unfortunately, it is not uncommon that certain (mostly internal) stakeholders foresee or in fact experience negative consequences from the changes that result from your project. In such cases, it will be hard to get their cooperation, even though you will certainly need it. Escalation to higher levels in the organization may then be the only way to proceed, even though the resulting relationship will be far from optimal. Stakeholder relationship management [Bour2009] is an effective way to counter problems with stakeholders.

This implies a continuous process of gaining and maintaining the support and commitment of stakeholders by engaging the right stakeholders at the right time and understanding and managing their expectations.

In order to engage stakeholders in the elicitation process, you must ensure that they know what the project is about and what their role within the project is. You have to understand

their needs and try to address these needs as far possible within your competencies in the project. While stakeholders deserve to be treated with respect, you may ask the same from the stakeholders, at least from those who are actively engaged in the project. This means that they should have time for you when you need them. They should give you the information that you ask for, as well as other information that they know to be relevant. Their feedback on your work products should be given timely and they should refrain from gossip about the project, etc.

Problems with stakeholders typically arise if the rights and obligations of the Requirements Engineer and the stakeholders with respect to the proposed system or the current project are not clear or if the respective needs are not sufficiently addressed. If problems are encountered, a kind of *stakeholder agreement* or *stakeholder contract* can help to provide all parties with the desired clarity. When this occurs within an organization, endorsement by senior management may add to the success of such an approach.

4.1.1.1 A Special Stakeholder: The User

Every system that we develop will have some interaction with certain users; why else would you develop it? Of course, when you are working on the requirements for an embedded technical subsystem, hidden inside some kind of complicated machinery, users will only interact with it indirectly through several layers of surrounding systems. In such cases, these users will not be your most important sources of requirements. However, in many systems, specific human beings will have a direct interface with the system: the (end–)users. Their acceptance of the system is vital to the success of the project, so they are your prime interest and will receive special attention during all Requirements Engineering.

There are two major categories of users:

- *Internal users* are directly related to the organization for which the system is being developed, such as staff, management, subcontractors. They are mostly limited in number, more or less known individually, and somehow involved in the project. It is relatively easy to contact them and they can be reached, influenced, and motivated through formal, existing channels.
- *External users* are outside the company, such as customers, business partners, civilians. Their number may be (very) large, in many cases they are not known individually, and they could be completely unaware of or indifferent to the project. You cannot influence them through formal channels. To get in contact with them, you may need to do special things to motivate them to participate, such as advertising, promising some reward, or giving them free access to a beta version. Forming a user panel or addressing the crowd (sometimes with payment) can be useful ways of involving these users.

Be aware that in addition to these regular categories, it can also be relevant to pay attention to *misusers*: people who intentionally try to interact with the system in a harmful way, such as hackers or competitors. It is rarely possible to contact or to influence them, but you can try to develop measures to discourage them, to keep them out, or to detect foreseeable attempts of misuse.

This categorization should not be considered very strictly. We can imagine projects in which users outside the company are small in number and can be reached easily, so they can be seen as *internal*. On the other hand, in big companies, the distance to the users can be so large that they should be treated more or less as *external*.

If you have a good insight into your user base, you should make a distinction between user roles. Separate roles will usually have different information needs, will use the system in their own way, and may have distinct access rights to functions and data—for instance, users who input data versus supervisors who check this input. In such cases, make sure that you include representatives of all relevant roles in the elicitation.

Often, especially at the beginning of elicitation efforts, such insight will be missing. Then, it is even more important to realize that there is no such thing as *The User*. *The User* is not an amorphous mass of identical humans but rather a collection of individuals, each of them with their own habits, wishes, and needs. When a system has thousands of users or more, of course you will not be able to fine tune the requirements to their individual needs. On the other hand, a *one size fits all* approach aiming for the *average* user might not work either.

In such cases, it helps to discern a number of user types or user groups that show certain, often behavioral similarities within the group as distinct from other groups. In practice, having some three to seven groups often works best. Then, as a Requirements Engineer, you will treat every group as a distinct source for requirements. A good technique is the use of *personas* [Hump2017]. Personas are fictitious individuals that describe typical user groups of the system with similar needs, goals, behaviors, or attitudes.

4.1.1.2 Personas

There are two major approaches to creating personas:

- Data-driven

  In this approach, personas are developed with marketing techniques, such as interviews, focus groups, and other ethnographic data collection techniques. Such personas are called *qualitative personas*. If personas are developed through statistical analysis of a large sample of your user base, they are called *quantitative personas*.

- Imagination

  As a cheaper and quicker alternative, personas may be developed by imagination—for instance, in a brainstorming session with the project team. We call them *ad hoc personas* or *proto-personas*. As a Requirements Engineer, you must be aware that ad hoc personas are based on imagination and assumptions. These assumptions must be checked and confirmed throughout the Requirements Engineering process.

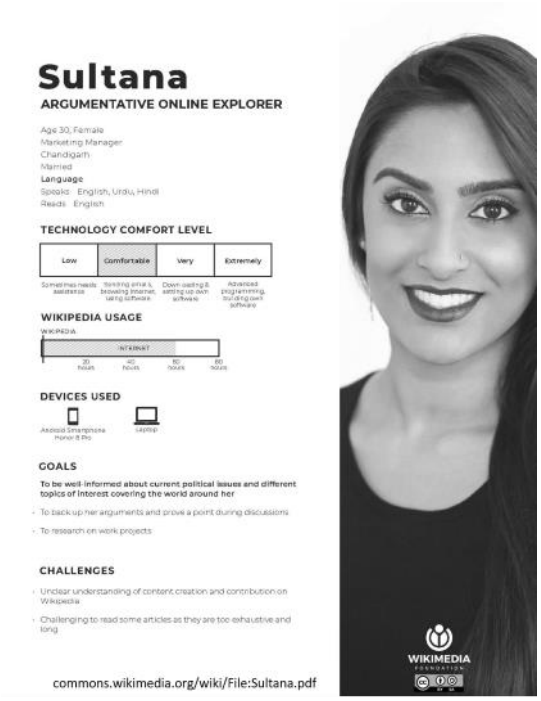


Figure 4.3 Persona example

Basically, persona descriptions contain information that is relevant in view of the development of the system at hand. Usually, this information will be *enriched* with additional data, such as name, address, hobbies, and a drawing or portrait picture.

This gives a human face to the abstract concept of persona. It may help you to understand that your work as a Requirements Engineer not only relates to facts but also to emotions. Figure 4.3 gives an example of a persona description.

If you use personas in your project, it may be useful to look for a few individuals that fit the persona descriptions and treat them as representatives of each group. In that case, you have real stakeholders with whom you can communicate. However, always remember that the group that such a stakeholder represents is an artificial one that does not exist in the real world but only in the context of the system or project.

4.1.2 Documents

Documents can be a major source for requirements too. As a Requirements Engineer, you often have to do a lot of reading, especially at the beginning of a project. All kinds of documents may be relevant: company-, domain- and project-related documents, product and process descriptions, legal and regulatory documentation, etc. As with stakeholders, you can make a distinction between *internal* and *external* documents. Internal documents can be easy to get but may be confidential and cannot be shared without explicit consent.

Often, you will need to sign a non-disclosure agreement before you receive access to them. External documents may be difficult to find but are usually public; if not, make sure that you are allowed to access and use them.

Documents can be a great way to find other sources. For instance, an internal process description may mention certain roles as being involved in that process, which in turn can lead you to new potential stakeholders. However, documents can also be direct sources for requirements, especially those that are easily overlooked or not regularly mentioned by stakeholders: constraints in standards, company guidelines, and other legal or regulatory documents; detailed requirements in procedures and work instructions; bright new ideas in marketing material from competitors. Studying documentation can help you to understand the context of the system to be developed, even by reading emails between people who took the initiative for the project. Reading about analogous solutions for problems and goals in other companies and domains can spark your imagination and show a feasible direction for your current project.

As a Requirements Engineer, you should be aware that a document is always related to some people: the author(s), the (target) audience, a manager responsible for or an auditor checking adherence to it, someone who pointed out its existence to you, etc. All those people may have a role as a stakeholder; it is up to you to find out whether or not this is the case. You should always check the validity and relevance of a document and you often need stakeholders to confirm this to you. If you were to derive requirements from an invalid or outdated document, the system developed from these requirements would probably fail.

Just like stakeholders, documents used as requirements sources must be managed. You can use a document list, comparable to the stakeholder list discussed above. All documents should be kept in some kind of common, indexed library with a unique identification to allow them to be referenced. Dates and version numbers are important to guard against working with outdated versions; you could check at regular intervals whether a newer version has been published and whether this influences the requirements. You should preferably work with final versions but in practice, you often have to deal with drafts, so you also have to record the status of documents. Keep old versions in an archive, because they may be important to understanding how a system and its requirements evolved. Setting up suitable and accurate management of the documents involved right from the start of a project will save you a lot of work later on, in Requirements Engineering, development, and deployment. It is a good starting point for establishing backward traceability, as discussed in Section 6.6.

4.1.3 Other Systems

You can also consider other systems as sources for requirements of the system you are interested in. Here, you can make a distinction between *internal* and *external* systems, just as in documentation and with the same considerations about access and confidentiality. Another distinction is that of *similar* systems versus *interfacing* systems.

Similar systems have certain functionalities in common with the system to be developed. They may be predecessor or legacy systems, competitor systems, comparable systems used in other organizations, etc. You often study them through their documentation but sometimes you can observe them in action or try them out as if they were a kind of prototype. You may be able to contact their users to learn more about the functionalities and

solutions of such systems. Predecessor or legacy systems are often a good source for identifying detailed (functional and quality) requirements and constraints.

However, be aware that (especially technical) constraints from the past may not be relevant anymore and may no longer restrict your current solution space.

Sometimes, a new system and a legacy system will coexist during a certain period, leading to additional requirements—for instance, with regard to data sharing. Competitor and comparable systems may be studied for their solution characteristics and can be a good source for identifying *delighters* (see Section 4.2.1).

Interfacing systems will have a direct relationship with the system for which you are developing the requirements. They will exchange data with your system as a source and/or a sink though some (synchronous or asynchronous, in real time or in batch) interface (see also Section 3.4.2 on system interfaces in context modeling). The correct configuration, content, and behavior of such an interface is often essential for ensuring that your system works, and you will therefore have to understand the system in detail. You can study interfacing systems through their documentation, but as every (technical) detail is important here, simulation or testing may be necessary. With regard to older or legacy systems in particular, you cannot trust their documentation to be up to date so you will need some proof. To understand an interface, you will also have to understand the context, functionality, and behavior of the interfacing system. It will be helpful if you can contact application managers, architects, or designers of such systems, especially if the interfacing system itself has to be updated to allow for the new interface. Also be aware that an interfacing system will itself have users; it may be interesting to consider these users as stakeholders in Alexander's *wider environment* of your own system.

4.2 Elicitation of Requirements

If we continue the analogy of water being drawn from a well, we are now at the point that we have found the well and we start pulling the rope to get the bucket full of the required water (or in this case requirements) to the surface. That is what we call *elicitation*: the effort expended by the Requirements Engineer to turn implicit desires, demands, wishes, needs, expectations—which until now were hidden in their sources—into explicit, understandable, recognizable, and verifiable requirements. Of course, we will have to use all wells to be complete and pull the rope in the right way to make sure that we get all the water to the surface. In Requirements Engineering terminology, we say that we should apply the right *elicitation techniques*.

A common categorization of elicitation techniques is the distinction between:

- Gathering techniques
- Design and idea-generating techniques

From these categories, you can select a wide range of elicitation techniques, each with their own characteristics. Figure 4.4 gives an overview of elicitation techniques in their categories and subcategories.

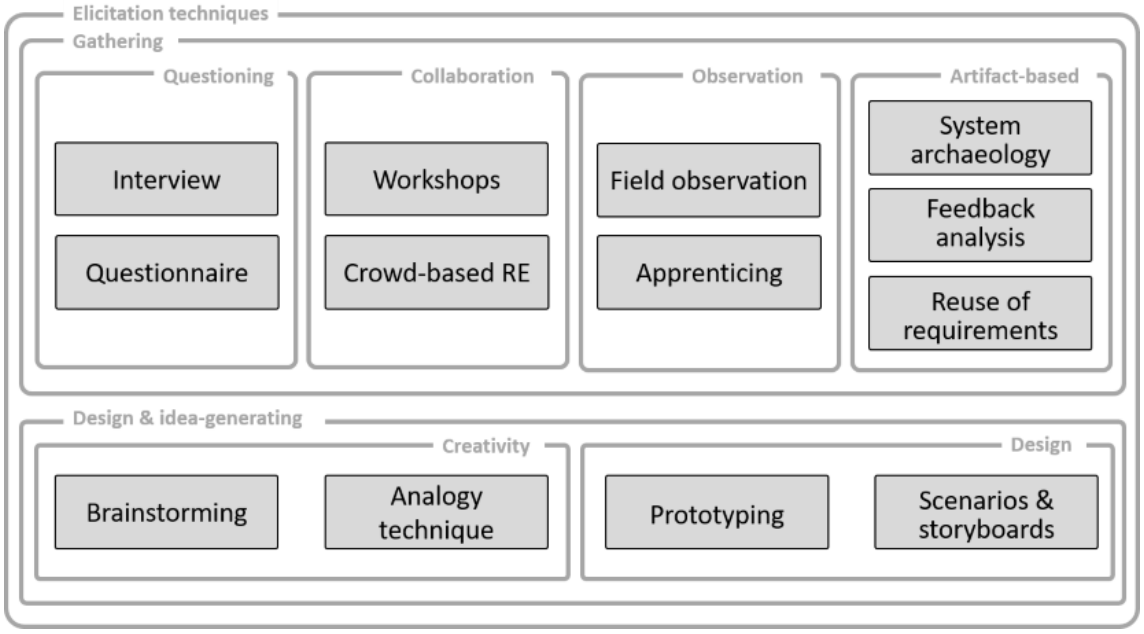


Figure 4.4 An overview of elicitation techniques

A critical key competence of the Requirements Engineer is the ability to choose the right (mix of) techniques under the given circumstances. Picking the right ones may depend on many factors, such as:

- Type of system

  A completely new innovative system will benefit more from design and idea-generating techniques, while a replacement system in a safety-critical environment may need questioning techniques and system archaeology.

- Software development life cycle model

  In a waterfall project, you may have planned for extensive techniques such as apprenticing or analogies, while in an agile environment, brainstorming, storyboarding, and prototyping may prevail.

- People involved

  For instance, field observation will probably not be appreciated in highly confidential businesses; a comprehensive survey may be preferred over a high number of individual interviews.

- Organizational setup

  A solid government organization needs a totally different approach to a young startup; a dispersed, highly decentralized company needs a different approach to a compact company with a single location.

The best results are usually achieved with a combination of different elicitation techniques. For a systematic approach to selecting them, see [CaDJ2014].

Elicitation techniques are—or at least, should be—able to detect all kinds of requirements. In Requirements Engineering practice, however, explicit *functional requirements* are often overrated, and the more implicit *quality requirements* and *constraints* get less attention.

This may result in a system that—with all functional requirements being fulfilled—does not perform, has poor usability, does not comply with architectural guidelines, or fails to meet certain other quality requirements or constraints, and consequently will not be accepted.

Stakeholders can be sources, but you will often find more information in documents. For the elicitation of *quality requirements*, applying a checklist based on the ISO 25010 standard [ISO25010] can help to detect and quantify them—for example, in preparing for an interview. *Constraints* can be found by considering possible restrictions of the solution space—for example, technical, architectural, legal, organizational, cultural, or environmental issues. Relevant documentation can often be identified through staff members.

4.2.1 The Kano Model

One of the major circumstances to consider in selecting an elicitation technique is the nature and the importance of a requirement that we are trying to uncover. To gain more insight into the nature of certain requirements, the Kano model [Verd2014] comes in handy. This model, shown in Figure 4.5, classifies features of a system into three categories:

- *Delighters* (synonyms: excitement factors, unconscious requirements)

  A delighter is a feature that customers are not aware of; that is why we call them *unconscious*. The customers do not ask for the feature because they do not know that it is possible in the system—for instance, a smartphone that can be turned into a beamer. At first, when the feature is new on the market, most customers will have their doubts about it, but when some early adopters have tried it out and start spreading the word, more and more people want to have it. If a delighter is absent, no one will complain; but when present, this can be a differentiating feature that attracts lots of customers.

- *Satisfiers* (synonyms: performance factors, conscious requirements)

  A satisfier is something that the customers explicitly ask for (hence *conscious* requirements). The more satisfiers you can put into your system, the higher the satisfaction of the customers will be. An example could be the number of lenses and video options in a modern smartphone. Because adding satisfying features usually also means higher costs, you will often need a kind of cost/benefit analysis to decide how many of them will be incorporated in the system.

- *Dissatisfiers* (synonyms: basic factors, subconscious requirements)

  A dissatisfier is also a feature that the customers do not ask for. Here, however, the reason for not asking for it is that the feature is so obvious (*subconscious*) that the customers cannot imagine it not being part of the system; these features are tacitly considered as must-haves. Imagine a smartphone without GPS. If a dissatisfier is included as a feature of a system, customers will not notice it because they think the

system cannot exist without it. However, if you overlook such a requirement and leave it out of the system, customers will be very upset and will refuse to use the system.

The Kano model looks at requirements from the perspective of the customer. It focuses on differentiating features, as opposed to expressed needs. With the Kano model in mind, you may find more requirements than when focusing only on the explicitly formulated needs from the stakeholders. As we will see later in this chapter, all categories can be linked to distinct elicitation techniques.

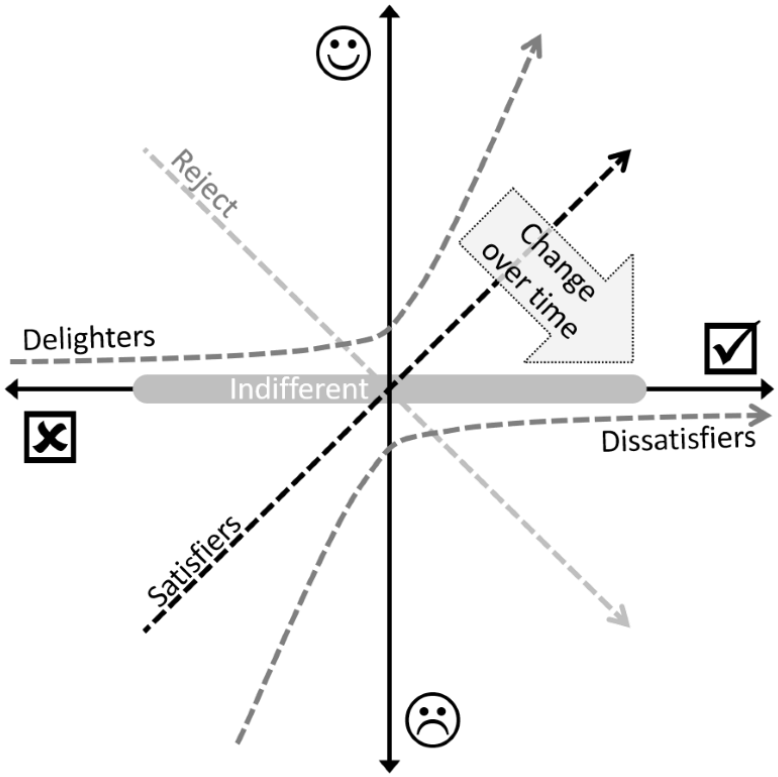


Figure 4.5 The Kano model

In fact, the original Kano model contains two more categories, the indifferent (or I don't care) and the reject (or I hate) requirements. These categories do not get much attention in most Requirements Engineering handbooks but can still be useful for you as a Requirements Engineer. Suppose, for instance, that developers want to add a certain feature to the system for technical reasons. If, after analysis, you find that the customers are indifferent to this feature, it is safe to include it in the system. However, if it turns out to be a reject requirement, you should tell the developers to look for a less harmful alternative, as implementing this requirement can turn out to be a costly mistake.

One interesting observation when working with the Kano model is that requirements tend to change over time. If someone introduces a new feature, there is no certainty about how the market will react to that feature. Sometimes, customers will be indifferent to it, and the feature will survive only if it does not increase the price of the product.

If customers reject it, the feature will probably be removed from the product as soon as possible. However, when (maybe initially a vanguard of) the customers like the feature, it will become a delighter, a unique selling point for which the customers are prepared to pay the price. As more and more customers discover, experience, and like this new feature, it will become a satisfier that is explicitly asked for. Gradually, when similar systems start to implement the same feature, customers may forget that systems did not originally include such a feature and will take it for granted, turning it into a dissatisfier. That is why many systems contain features that the users consider as indispensable without knowing why and thus without explicitly asking for them.

A good example is the camera function on cell phones, for which this process took less than 20 years. The first time a camera was introduced as part of a cell phone, most customers were puzzled: no one had asked for this feature and most customers thought "If I want to take a picture, I need a camera." However, some early adopters tried it out and discovered the convenience of taking pictures without a dedicated camera and being able to share them instantly with other people without making a print. They liked the camera feature as a delighter and all brands started to implement it in their phones, turning it into a satisfier: the better the pictures were, the more satisfied the user was. Nowadays, when buying a new cell phone, everybody takes for granted that it will have a camera function so it has become a dissatisfier: "If I can't take a picture with this cell phone, it is useless."

How can you categorize a specific feature? You use the technique of Kano analysis. For a specific feature, you ask two questions to a representative group of potential users: (1) "What would you feel if this feature were present in the system?" and (2) "What would you feel if this feature were absent from the system?" You let them score the answers on a 5-point scale between "I love it" and "I hate it" and then plot the average answer on the Kano analysis matrix as shown in Figure 4.5. The cell that comes up gives you the Kano classification for the feature.

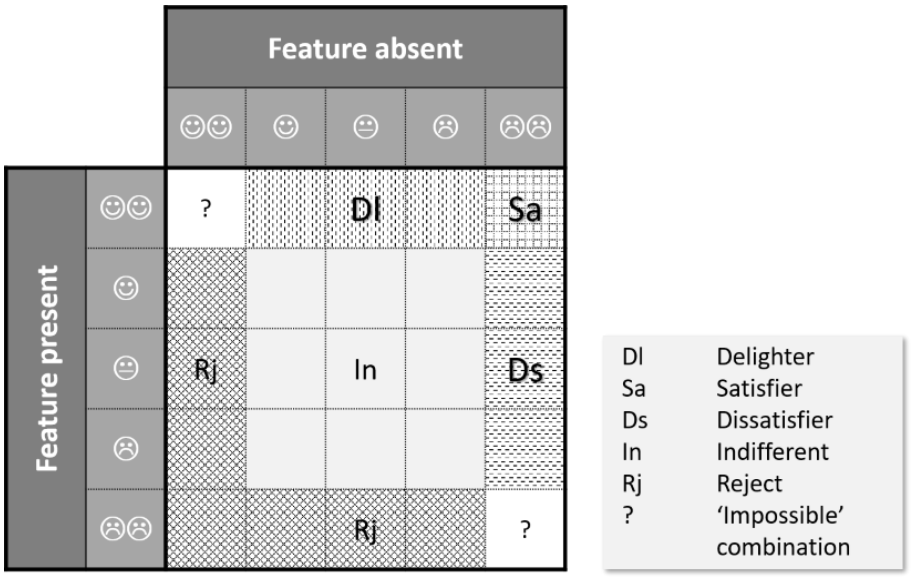


Figure 4.6 Kano analysis matrix

The next question is: why bother with Kano analysis in requirements elicitation?

As we explain in the following sections, you will need different techniques to find these different categories of features. By themselves, stakeholders will mainly talk about their satisfiers—their conscious requirements that they explicitly ask for. It is much more difficult to detect the other categories but fortunately, there are several useful techniques for doing so.

4.2.2 Gathering Techniques

With *gathering techniques,* you examine the different sources that you have identified and elicit the requirements from there. These established techniques have been commonly used throughout Requirements Engineering and predominantly yield satisfiers and dissatisfiers.

Gathering techniques can be further subdivided into four categories:

- Questioning techniques
- Collaboration techniques
- Observation techniques
- Artifact-based techniques

Questioning techniques are always used in an interaction with stakeholders. The Requirements Engineer poses appropriate questions to the stakeholders in order to let the stakeholder do the thinking and to receive answers from which requirements can be derived.

Examples of questioning techniques are:

- Interviews

  Due to their flexibility, *interviews* are probably one of the most frequently used elicitation techniques. They do not require specific tools and can be used to elicit high-level requirements as well as very specific ones. Usually, an interview is a one-to-one session between a Requirements Engineer (interviewer) and an individual stakeholder (interviewee), but a small group of interviewees is also an option. Typically, requirements elicited with an interview are satisfiers, as the interviewee voices conscious information. The interview technique is not overly complicated and most people have a good understanding of what it is. However, you need clear goals and good preparation to obtain useful results. Interviews can reveal detailed information and offer flexibility based on the answers given. They are rather time-consuming, so this technique is less appropriate when you want to reach large numbers of stakeholders.

- Questionnaires

  With a *questionnaire*, a larger group of stakeholders is asked to answer—orally, in writing, or on a web page—the same set of questions, which are presented in a structured way. Quantitative questionnaires are used to confirm hypotheses or previously elicited requirements. They use closed-ended questions (only predefined answers allowed) and can therefore be evaluated quickly and deliver statistical

information. On the other hand, qualitative questionnaires use open-ended questions and can find new requirements. They tend to deliver complex results and are thus usually more time-consuming to prepare and to evaluate. In general, questionnaires are a preferred technique for large groups. Be aware, however, that designing a good questionnaire involves quite a lot of effort. A questionnaire is often the next step after obtaining a preliminary idea based on a series of interviews in order to validate these ideas within a larger group.

In the category of *collaboration techniques,* we find all kinds of collaboration between the Requirements Engineer and other people (stakeholders, experts, users, customers, etc.). Some examples are:

- Workshops

  Workshop is an umbrella term for group-oriented techniques, ranging from small informal meetings to organized events with several dozen or even hundreds of stakeholders. A nice definition is as follows: "A requirements workshop is a structured meeting in which a carefully selected group of stakeholders and content experts work together to define, create, refine, and reach a closure on deliverables (such as models and documents) that represent user requirements" [Gott2002]. With a workshop, you can get a good global insight in a short time because you use the interaction between the participants. If you need more detail, follow-up interviews or questionnaires are appropriate. Workshops can serve as a gathering technique but they can also be used in creativity techniques (see Section 4.2.3).

- Crowd-based Requirements Engineering

  In *crowd-based* (also known as platform-based) Requirements Engineering (see [GreA2017]), elicitation is turned into a participatory effort with a crowd of stakeholders, in particular the users, leading to more accurate requirements and ultimately better software. The power of the crowd lies in the diversity of talents and expertise available within the crowd. As the amount of data obtained from the crowd will be large, an automated platform for processing this data is essential. This platform should offer community-oriented features that support collaboration and knowledge sharing and foster the engagement of larger groups of stakeholders in the collection, analysis, and development of software requirements, as well as validation and prioritization of these requirements in a dynamic, user-driven way.

Observation techniques are also applied in relation to stakeholders. The stakeholders are observed while they are engaged in their normal (business) processes in their usual context without direct interference from the Requirements Engineer. Observation techniques are particularly useful for identifying dissatisfiers. You may observe peculiar activities, sequences, data, etc. that are so common to the stakeholders that they do not mention them, and these aspects thus do not easily come to light in gathering techniques.

Common forms of observation techniques are:

- Field observation

  During *field observation*, the Requirements Engineer watches (mostly) end users in their environment while these users perform the activities for which a system is to be developed. Field observation is typically used in situations where interaction would distract the users or would interfere with the process itself and potentially falsify results. It can even be applied without informing the subjects observed, e.g. by sitting with other patients in a dentist's waiting room to observe their behavior. With field observation, you will be able to detect (often detailed) requirements that would not easily be found with other techniques—for instance, because actions and behaviors are too complicated to put into words.
  Be aware that field observation requires a lot of preparation, a sharp eye, and lots of time. Video is quite helpful for capturing stakeholder behavior. It can be used in conjunction with direct field observation and may even replace it in situations where the actual presence of the Requirements Engineer is not allowed or desired. Video offers the possibility of postprocessing to allow for detailed investigation of acts and proceedings that are difficult to observe.

- Apprenticing

  Apprenticing differs from field observation in that it is participatory. In apprenticing, the Requirements Engineer (*apprentice*) does a kind of internship in the environment in which the system at hand will be used (or is already in use) and experienced users (*masters*) teach the apprentice how things work. The apprentice participates but does not interfere; they act like a novice in the field and are allowed to make mistakes and ask "dumb" questions. The aim is to create a deep understanding of the domain, the business, and the processes before the actual elicitation of the requirements starts. A follow-up with interviews and questionnaires will often be required to verify the initial ideas. The resulting requirements can subsequently be documented and validated. An optimal duration for such an internship depends on many different factors (e.g., complexity of the process, repetitiveness, time availability of master and apprentice) but usually varies between one day and several weeks. Be aware that apprenticing may be difficult or impossible to organize in certain domains, such as medicine, aviation, or the military.

Artifact-based techniques do not use stakeholders (directly) but rather work products such as documents and systems, or even images, audio and video files, as sources for requirements. These techniques can find (sometimes very detailed) satisfiers and dissatisfiers. It is usually a time-consuming task to examine (often poorly structured, outdated, or partly irrelevant) work products in detail. Nonetheless, artifact-based techniques can be useful, particularly when stakeholders are not readily available.

A few examples of artifact-based techniques are:

- System archaeology

  In *system archaeology*, requirements are extracted from existing systems—such as legacy systems, competitor systems, or even analogous systems—by analyzing their documentation (designs, manuals) or implementation (code, comments, scripts, user stories, test cases). This technique is mainly used if an existing system has been used for many years and is now to be replaced by a new system for whatever reason; the new system has to cover the same functionality as the old one, or at least certain parts of it. System archaeology often takes a lot of time but may reveal detailed requirements and constraints that are not easily detected otherwise. However, you will need extra time to check, through other channels, whether or not these requirements are still valid and relevant.

- Feedback analysis

  There are many ways to collect *feedback* from (potential) users and customers, be it on an existing system or on a prototype. Feedback data may be structured (e.g., a 5-star rating in an app store) or unstructured (like review comments). It may be gathered via web surveys and contact forms, during beta or A/B testing, on social media, or even as customer remarks received in a call center. Often, the amount of data is quite large, and analysis will be time-consuming. However, the feedback can be very useful for gaining insight into the user's *pains and gains*. Negative scores and critical remarks will help you to detect unnoticed dissatisfiers. Positive scores and compliments will give you additional information about satisfiers. Occasionally, comments may even contain innovative ideas that can be turned into delighters. Feedback analysis can thus result in adjustment of existing requirements but also to the discovery of new ones.

- Reuse of requirements

  Many organizations already have a large collection of requirements that have been elicited and elaborated in the past for previous systems. Many of these requirements may be applicable for a new system too, especially requirements that have been derived from an overarching domain model. Therefore, *reuse* of existing requirements can save lots of time and money because you can skip their elicitation. However, this works only if this collection of existing requirements is up to date, managed effectively, easily available, and documented extensively, which unfortunately is not often the case. Even if reuse is feasible, be aware that you still need to validate with the stakeholders whether these reusable requirements are relevant and valid in the new situation, be it directly or with some adjustments.

4.2.3 Design and Idea-Generating Techniques

In the past, Requirements Engineering has focused on gathering and documenting the necessary requirements from all relevant stakeholders by applying gathering techniques as introduced in the previous section. The growing influence of software as an innovation driver

in many businesses is now increasingly demanding a new positioning of Requirements Engineering as a creative, problem-solving activity. This involves the application of other techniques that no longer consider stakeholders (and their documents and systems) the one and only source of requirements. Innovative systems need new, maybe disruptive features that the current stakeholders cannot imagine (yet).

Design and idea-generating techniques have emerged to fulfill this need. These techniques are intended to stimulate creativity, mostly within teams, for the generation of ideas and may provide additional ways to elaborate a given idea. These techniques may yield new and innovative requirements that are often delighters. Many diverse techniques exist within this broad category, some remarkably simple, others quite elaborate. We will look at a few examples from two subcategories:

- Creativity techniques
- Design techniques

In addition, we will look at the emerging field of *design thinking*.

Creativity techniques stimulate creativity in order to find or to create new requirements that cannot be gathered directly from the stakeholders because the stakeholders are not aware of the feasibility of certain new features or (technical) innovations. These techniques are usually applied within diverse, multi-disciplinary teams of IT staff such as analysts, Requirements Engineers, developers, testers, product owners, application managers, etc., with or without business representatives, users, clients, and other stakeholders. The techniques stimulate out-of-the-box and borderless thinking and elaboration of each other's ideas. Unfortunately, none of them guarantee success in generating creative results as several mechanisms in our brain have to come together to enable creative ideas.

> An obvious example where creativity techniques are important is the games industry. You can of course ask gamers for their requirements with a gathering technique and you will learn what gamers like or dislike about the current games. However, to develop a successful game, you need to surprise the gamers with something new; you have to discover their delighters. That is exactly where creativity techniques fit in.

Several preconditions have been identified as important factors for creativity to emerge:

- Chance—and therefore time—for an idea to come up
- Knowledge of the subject matter, which raises the odds for an idea that makes the difference
- Motivation, as our brain can only be creative if there is a direct benefit for its owner
- Safety and security, as useless ideas must not have negative consequences

Two examples of creativity techniques are presented here:

- Brainstorming

  Brainstorming (see [Osbo1948]) supports the development of new ideas for a given question or problem. As with most creativity techniques, the crucial point of

brainstorming is to defer judgment by separating the finding of ideas from the analysis of ideas. Some general guidelines for brainstorming include:
Quantity prevails over quality.
Free association and visionary thinking are explicitly desired.
Taking on and combining expressed ideas is allowed and desired.
Criticizing other participants' ideas is forbidden even if an idea seems to be absurd.
After a brainstorming session, the ideas that have emerged are categorized, assessed, and prioritized. Selected ideas then serve as input for further elicitation.

- Analogy technique

  The *analogy technique* (see [Robe2001]) helps with the development of ideas for critical and complex topics. It uses analogies to support thinking and the generation of ideas. Its success or failure is influenced mainly by the selection of a proper analogy for the given problem. The selected analogy can be close to (e.g., the same problem in another business) or distant from (e.g., comparing an organization with a living organism) the original problem. The application of the analogy technique consists of two steps:
  Elaborate the aspects of the selected analogy in detail without referring to the original problem.
  Transfer all identified aspects of the analogy back to the original problem.
  The resulting concepts and ideas will then be a starting point for additional elicitation.

Design techniques help explore and elaborate ideas generated with creativity techniques and also help clarify and concretize vague stakeholder needs. They heavily rely on visual or tangible artifacts, team cooperation and customer feedback.

Popular techniques in this category include:

- Prototyping

  By *prototype* (in relation to elicitation; see also Section 0 for more information), we mean a kind of intermediate work product that is created or released to generate feedback. Prototypes can range from simple paper sketches to working pre-release versions of a system. They allow future users to experiment with the system in a more or less tangible way and to investigate certain, as yet unclear, characteristics during Requirements Engineering and before the actual implementation. As we will see in the section on validation (4.4.2), prototypes are primarily used for checking that previously defined requirements have been implemented correctly. However, with proper guidance of the users and analysis of their feedback, this technique can also be used to derive new requirements. It may be particularly useful for detecting non-functional requirements, dissatisfiers and constraints, or whatever other characteristics that cannot easily be understood or defined up front in models and documentation.

- Scenarios and storyboards

  The word *scenario* stems from the theater, where it is used to refer to an outline of a play, opera, or similar, indicating a sequence of scenes with their characters. In IT, we

use this term to describe a flow of actions for a system, including the users involved (who we usually call actors here). Through scenarios, you can explore alternative ways of realizing a process in a system. Because of their lightweight structure, they are easy to develop and can be changed rapidly. In the same way as for prototypes, scenarios and storyboards can be applied in both (early) elicitation and (later) validation of requirements.
Scenarios can be documented in a written or a visual form. The visual form of a scenario is called a *storyboard*.
A storyboard is typically a kind of comic strip with a series of panels that show the interaction of certain personas with the system. See Figure 4.7 for an example.
Scenarios and storyboards are useful for early elaboration of ideas in terms of processes and activities.

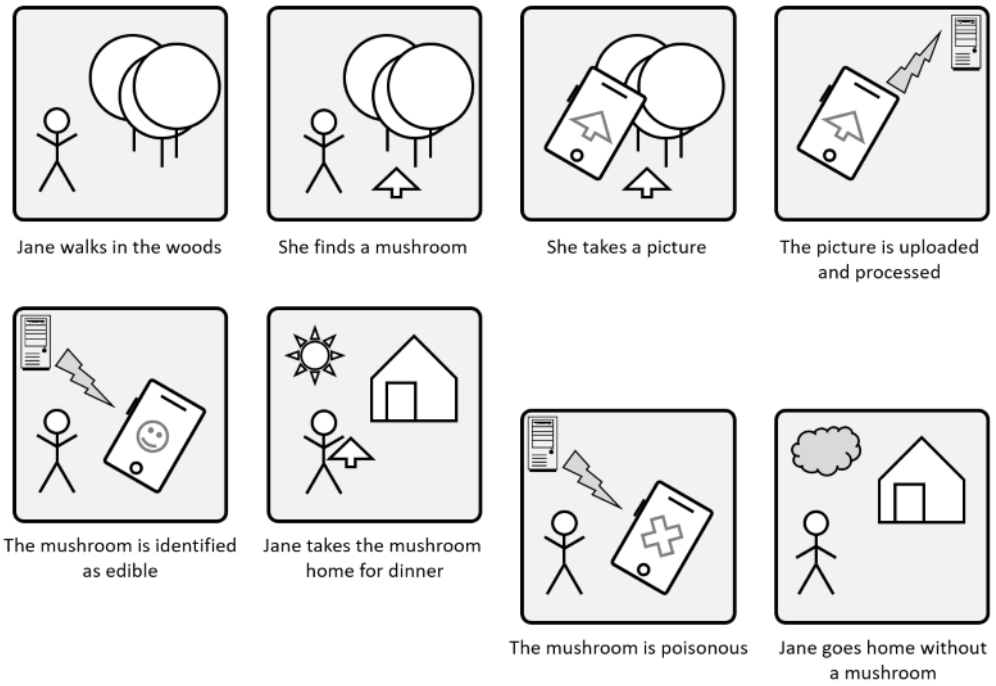


Figure 4.7 Example of a storyboard

Design thinking is not so much a technique but rather a concept, an attitude, a philosophy, a family of processes, and often a toolbox full of techniques. The focus is on innovation and problem solving. Several variants of design thinking exist, mostly using lightweight, visual, and agile techniques. Two basic principles can be found in all variants:

- Empathy

  The first step for design thinkers is to find the real problem behind the given problem. They try to understand what stakeholders really think, feel, and do when they interact with a system. Therefore, we often refer to design thinking as human-centered

design. Personas, empathy mapping, and customer co-creation are common techniques to this end.

- Creativity

  A common characteristic of design thinking is the *diamond*: the alternation of divergent and convergent thinking. Divergent thinking aims at exploring an issue more widely and deeply, generating lots of different ideas, and convergent thinking focuses, selects, prunes, combines these ideas into a single final delivery. A basic pattern, the *double diamond* model, is shown in Figure 4.8 (see [DeCo2007]).

A detailed treatment of design thinking is beyond the scope of this Foundation Level Handbook.

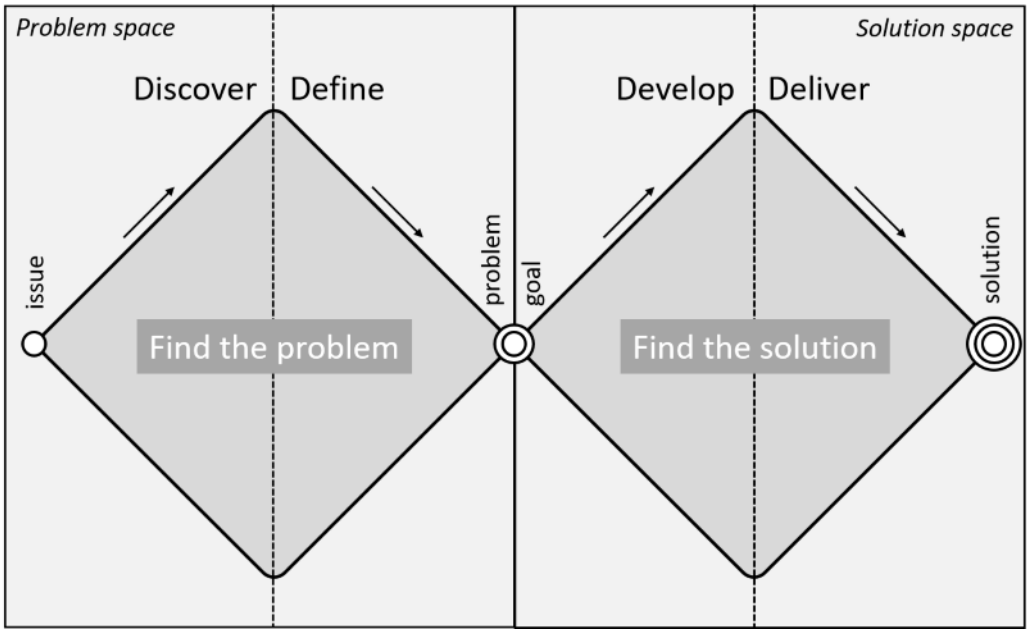


Figure 4.8 The double diamond

4.3 Resolving Conflicts regarding Requirements

During elicitation, you gather a broad collection of requirements from different sources, with different techniques, and at different levels of abstraction and detail. The elicitation techniques that you use do not guarantee by themselves that this collection as a whole forms a single, consistent, agreed upon set of requirements that captures the essence of the system. Both during and after elicitation of a set of requirements for a certain system, you may find out that some of the requirements are conflicting: they may be inconsistent, incompatible, contradictory. It might be that requirements conflict with each other (e.g., "all text must be black on white" versus "all error messages must be red") or that some stakeholders have a different opinion about the same requirement (e.g., "all error messages must be red" versus "user error messages must be red, all other error messages blue"). As we cannot develop a (specific part of a) system based on conflicting requirements, the conflicts must be resolved before development can start. As a Requirements Engineer, you are the

one who should make sure that all stakeholders arrive at a shared understanding (see Chapter 2, Principle 3) of the complete set of requirements as far as they are relevant to them and that they agree on this set.

But what is a conflict? A conflict is a certain disagreement between people: "An interaction between agents (individuals, groups, organizations, etc.), where at least one agent perceives incompatibilities between her thinking/ideas/perceptions and/or feelings and/or will and that of the other agent (or agents), and feels restricted by the other's action" [Glas1999]. In a requirements conflict, two or more stakeholders have a different or even contradictory opinion regarding a certain requirement or their requirements cannot be implemented in a certain system at the same time; see Figure 4.9.

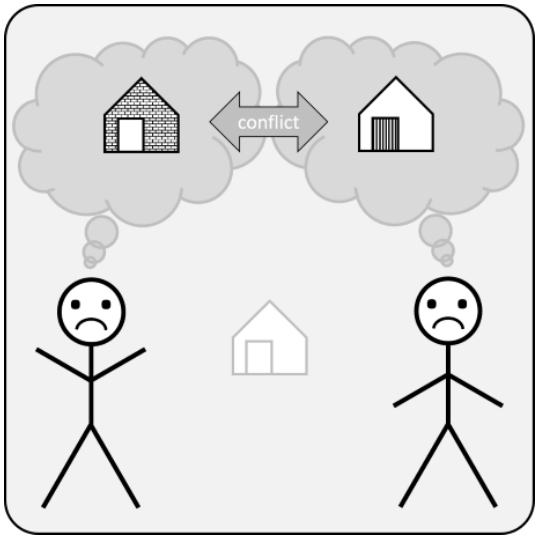


Figure 4.9 A requirements conflict

Dealing with requirements conflicts can be difficult, painful, and time-consuming, especially when personal issues are involved. However, denying or ignoring conflicts is not an option, so the Requirements Engineer must actively search for ways to resolve them. At the end, all stakeholders must understand and agree upon all requirements that are relevant to them. If some stakeholders do not agree, this situation must be recognized as a conflict that must be resolved accordingly.

4.3.1 How Do You Resolve a Requirements Conflict?

To resolve a requirements conflict properly, the following steps should be followed:

- Conflict identification

  We often have conflicts in our everyday life. They give us an unpleasant feeling, so a common strategy is simply to avoid, ignore, or deny them. That may make conflicts hard to find. Most of them tend to be hidden and can only be detected by careful observation. There are many indicators that you can pay attention to, both in communication and in documentation:

In communication, you may observe behavior such as denial, indifference, pedantry, continuously asking for more details, deliberately incorrect interpretations, concealment, or delegation.
In documentation, you may find things such as contradictory statements by stakeholders, conflicting results from analysis of documents or systems, inconsistencies across different levels of detail, and inconsistent use of terms.
If you observe such indicators, this does not necessarily mean that there is a requirements conflict, but you should certainly be suspicious. Thorough discussion with the stakeholders can then bring a hidden conflict to the surface.

- Conflict analysis

  Once a conflict has been identified, the Requirements Engineer has to first clarify whether this conflict is a requirements conflict or not. After all, a requirements conflict is the primary responsibility of the Requirements Engineer; other conflicts can be resolved by other participants, such as a department manager or a team lead. The Requirements Engineer should fully understand the nature of the requirements conflict before attempting to resolve it. This means that you will have to collect more information about the conflict itself and the stakeholders involved.
  Many aspects deserve attention:

  - *Subject matter*: the scope, the problem, or the real issue behind the conflict.
  - *Affected requirements*: which specific requirements are affected?
  - *Stakeholders involved*: who disagrees with whom about what?
  - *Opinions* of the stakeholders: let them make their point as clearly as possible so that all conflicting parties understand the underlying issue.
  - The *cause* of the conflict: what is the reason behind the difference in opinions?
  - The *history* of the conflict: what has happened before that influences these opinions now?
  - *Consequences*: the estimated costs and risks associated both with resolving the conflict or not resolving it.
  - *Project constraints*: personal, organizational, content-specific, or domain-specific constraints may determine the solution space.
  - Analyzing this information will help you to recognize the type of conflict (for more information, see Section 4.3.2) and will indicate ways to resolve it.
- Conflict resolution

  Once an in-depth understanding of the nature of the requirements conflict, the attitude of the stakeholders involved, and the project constraints has been reached, the Requirements Engineer will select a suitable resolution technique. Many techniques can be used, as explained in Section 4.3.3. The first step should always be to get the chosen technique accepted by the stakeholders involved before applying it. If some stakeholders do not agree up front with the application of a certain technique, they certainly will not accept the outcome of it, so at the end, the conflict will not be resolved. In principle, the Requirements Engineer is not one of the stakeholders involved, so you can and should apply the selected resolution

techniques in an objective, strictly neutral way, and welcome any outcome that results from applying the technique.

- Documentation of conflict resolution. Conflict resolution may influence the requirements in a way that is not obvious for someone who was not involved in the conflict. The resulting set of requirements may seem illogical or inefficient. Therefore, the conflict resolution should be properly documented and communicated with regard to aspects such as the following:
  - Assumptions concerning the conflict and its resolution
  - Potential alternatives considered
  - Constraints influencing the chosen technique and/or resolution
  - The way the conflict was resolved, including reasons for the chosen resolution
  - Decision-makers and other contributors
  - If you do not document the resolution, after a while, stakeholders may simply forget or ignore the decisions that have been taken. And later in the project, developers may not understand the rationale behind a particular system design and may implement it in a different way.

You do not need to be afraid of requirements conflicts, as they will always occur. This should not be a surprise to you; in fact, you should be troubled if you do not detect any conflicts. They are quite common, so if you do not find them, you have probably missed some. But never ignore them. If you do not resolve all requirements conflicts that you notice right away, they will pop up later in the development process. And as Barry Boehm [Boeh1981] already found out a long time ago, the later you discover a problem, the more expensive it will be to solve it.

4.3.2 Conflict Types

To achieve a better understanding of the nature of a conflict, it is useful to distinguish between different conflict types. This helps in selecting proper resolution techniques.

We discern six types of conflict:

- Subject matter conflict

  A *subject matter* conflict occurs when the conflicting parties really have different factual needs, mostly caused by the intended use of the system in different environments. A good example is a system that is to be used in different countries, each with their own legislation. It may be difficult to resolve such a conflict because the underlying facts cannot be changed. The first thing to do then is to analyze and document these facts in detail and to have the conflicting parties agree on the exact nature of the conflict.

- Data conflict

  A *data conflict* is present when some parties refer to inconsistent data from different sources or interpret the same data in a different way. This may be due to poor communication, missing background data, cultural differences, existing prejudices,

etc. Estimates in particular, such as future sales, can easily generate a data conflict as they are often based on assumptions. Detecting a data conflict is not easy, because as a Requirements Engineer, you may think that your own sources are right and your own interpretation is self-evident. Due to this bias, you often suspect another conflict type at first.
Understanding how people can come to a different interpretation requires a lot of empathy. Communication—over and over again—is key for both detecting and resolving this type of conflict.

- Interest conflict

  An *interest conflict* is based on different positions of the conflicting parties, formed by personal goals, goals related to a group, or goals related to a role. You should understand the concerns and needs of the stakeholders involved before you can resolve this type of conflict. However, keep in mind that in the case of personal interests, stakeholders often do not reveal their true motives and they put forward seemingly factual but essentially artificial arguments. If a discussion is about an interest conflict, you can observe the conflict parties trying to convince each other to follow their arguments and understand the needs of the role or group. Resolution may benefit from identifying and strengthening shared interests. Working on a mutual understanding about the gains and pains of both parties can be a starting point for finding a solution.

- Value conflict

  A *value conflict* is based on differences in values and principles of the stakeholders involved. Compared to an interest conflict, a value conflict is more individual and related to global and long-term perspectives. Values are more stable than interests and rarely change in the short term. If a value conflict is the reason for a discussion, the conflict parties will emphasize why their arguments are important from their point of view, revealing their inner values and principles. They tend to insist on their arguments and are unwilling to give up. To resolve such conflicts, look for higher values that unite the parties. Value conflicts are notoriously difficult to resolve and achieving mutual understanding and recognition of each other's principles is the best you can get.

- Relationship conflict

  A *relationship conflict* is usually based on negative experiences with another party in the past, or in comparable situations with similar people. Often, emotions and miscommunication are involved, which makes the conflict a lot more difficult to solve. Conflict parties misuse discussions on requirements to express their anger with the behavior of each other, forgetting about facts, figures, and fairness. Bringing the discussion back to requirements will rarely help; sometimes, uniting parties around a higher value is successful. In most cases, you will have to escalate the issue to other stakeholders or a higher level of authority; exchanging people is a potential resolution. Be aware that a relationship conflict often co-occurs with other conflict types—for

instance, an interest conflict. Analyzing the root cause and solving the other conflict type may then be the best way to improve the relationship.

- Structural conflict

  We call a conflict *structural* when it involves inequality of power, competition over limited resources, or structural dependencies between parties. The resulting imbalance (often perceived by only one of the parties) causes problems in communication and decision making. Another reason for such conflicts may be restrictions on resources or dependencies on work products to be delivered by another party. Parties may use the discussion on requirements to either change or preserve the status quo. Hierarchy may be misused to push through decisions. For structural conflicts too, escalating the issue to other stakeholders or a higher level of authority is often necessary.

Most requirements conflicts can be categorized as either a subject matter, data, interest, or value conflict. Relationship and structural conflicts are often not directly related to requirements and therefore the Requirements Engineer may not be the appropriate party to resolve them. However, in reality, most conflicts fall into more than one category as different causes interact. Therefore, it is advisable to pay attention to all kinds of conflicts, even if the solution is not within your own responsibility. If someone else should resolve the conflict, make sure that it happens; as long as a conflict is not resolved, it will continue to have a negative impact on your work as a Requirements Engineer.

4.3.3 Conflict Resolution Techniques

Depending on the type and the context (stakeholders, constraints, etc.) of a conflict, a proper resolution technique is selected. Commonly used techniques include [PoRu2015]:

Agreement

An *agreement* results from a discussion between the stakeholders involved, to be continued until they completely understand each other's positions and agree to a certain option preferred by all parties. It can be very time-consuming, especially when multiple parties are involved. If successful, it will provide additional motivation to the stakeholders, so the result has a good chance of being long lasting. Striving to reach an agreement is common in data conflicts. If this technique is unsuccessful within an acceptable timeframe, other techniques can be used thereafter.

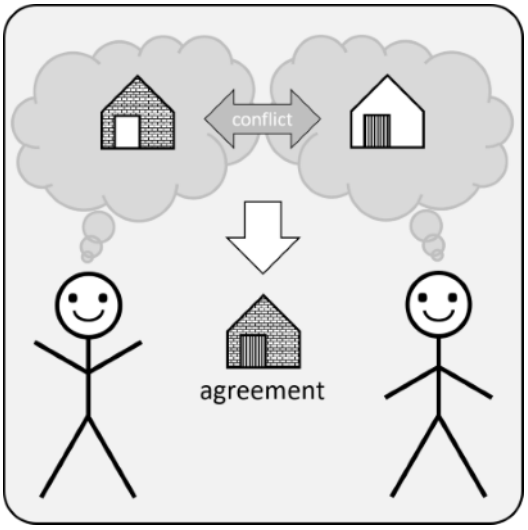


Figure 4.10 Agreement

Compromise

A *compromise* is quite similar to an agreement. Here, however, stakeholders agree on an option that is not their preference but that they can live with because accepting the compromise is considered better than continuing the conflict. Therefore, a compromise can also be long lasting. The compromise may contain new elements that were not present in the original preferences of the stakeholders and that may have been introduced by the Requirements Engineer. A good compromise is an alternative in which all parties feel comfortable with the balance of giving up things and getting something else in return. A compromise is often next in line if an agreement cannot be reached in time. It is suitable for subject matter conflicts and may also work for interest and structural conflicts.

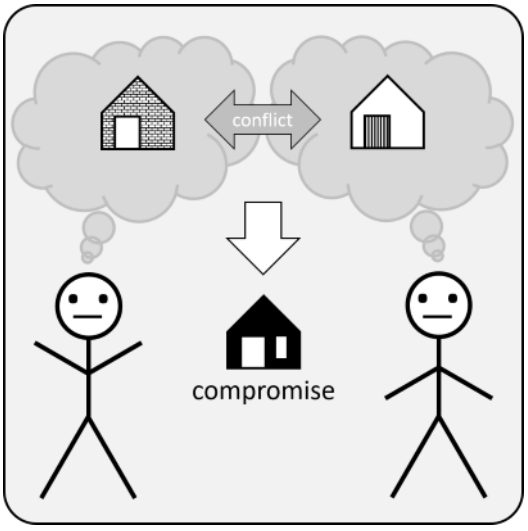


Figure 4.11 Compromise

Voting

Voting works best when a relatively simple choice has to be made between a clear set of conflicting requirements. Stakeholders that participate in the voting (usually not only the conflicting parties but all stakeholders involved) should fully understand the alternatives and the consequences of their vote. In order to avoid influences from dependencies or an imbalance of power, voting is best done anonymously and with a neutral moderator. The voting procedure itself should be agreed upon between the stakeholders before the actual voting. Voting is a quick and easy means for conflict resolution but the party that loses the vote will be disappointed and may need attention. Voting can work for most conflict types and may be a good way to solve subject matter and interest conflicts.

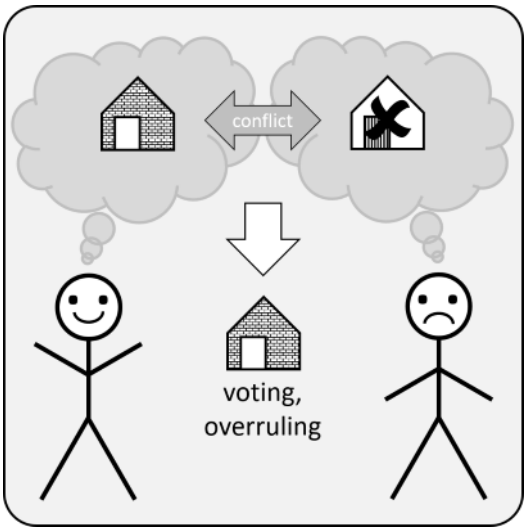


Figure 4.12 Voting

Overruling

If an agreement or a compromise cannot be reached and at least one of the conflicting parties refuses to participate in voting, *overruling* may be an option. It is often applied under pressure, when there is not enough time to use more convenient techniques. Usually, overruling is done by transferring the choice between conflicting requirements to a decision maker who is higher in authority or hierarchy than all conflicting parties and has enough power to have the decision be implemented. Therefore, it is a good way to solve interest and structural conflicts. In this situation, it is particularly important that the decision maker fully understands the alternatives, the position of the conflicting parties, and the consequences of the decision. A variant of overruling is to outsource the decision to a third party—for instance, an external expert. In that case, it is important to first get an agreement between the stakeholders on the decision maker. As with voting, you may need to pay attention to the *loser*.

Definition of variants

Definition of variants is often considered for subject matter, interest, and value conflicts. We have seen that we cannot implement conflicting requirements in one and the same system.

Definition of variants means that we build separate solutions for all conflicting requirements. This is usually implemented by developing a system that can be configured through parameters to exhibit the desired features. This may seem like a perfect solution but it comes at a price: it takes a lot of time to define the solution and a growing complexity (as well as additional costs) is introduced into the system, both for development and during operations and maintenance. This technique is therefore feasible only if enough time and budget are available.

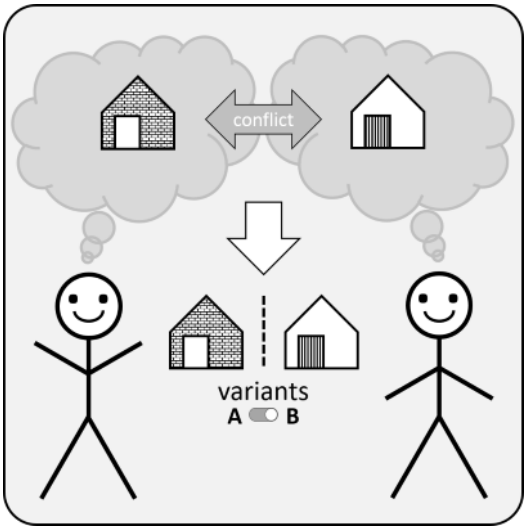


Figure 4.13 Definition of variants

Auxiliary techniques

In addition, there are several *auxiliary techniques* that are not usually used on their own but rather to assist the above-mentioned techniques.

In *Consider-All-Facts* (CAF), you consider alternative solutions for a number of predefined criteria—for example, cost, time, risk, available resources. Weighing these criteria can provide more clarity about the pros and cons of the alternatives and help to identify the *best* alternative.

Plus-Minus-Interesting (PMI, see [DeBo2005]) is a brainstorming and decision-making tool. It encourages the examination of ideas and concepts from more than one perspective and is therefore valuable for conflict resolution. In PMI, the participants (usually all stakeholders involved) first identify all positive aspects (plus) of the alternatives, then the negatives (minus), and finally the interesting points, things that need further investigation. The alternative with the most pluses and the fewest minuses is the preferred alternative.

In fact, both CAF and PMI are variants of the *decision matrix*, a *methodical* approach for conflict resolution. The conflicting requirements are assessed based on a (larger) number of criteria, after which, scores on these aspects are used to calculate a (weighted) final score for the alternatives. The *highest* score then wins, like *Alternative 1* in the example of Tabelle 4.1 below. In fact, prioritization (see Section 6.8) is then used as a resolution technique. As stated earlier, these techniques are usually seen as auxiliary: they create more

insight into the alternatives and thus help with the chosen resolution technique. They can even be used as a single technique if all stakeholders involved agree to accept the outcome.

Tabelle 4.1: Example of a decision matrix

		Alternative 1: iOS only		Alternative 2: Android & iOS	
Criterion	Weight	Score	Weighted	Score	Weighted
Cust. base	2	3	6	4	8
Dev. cost	1	3	3	2	2
T.t. market	3	4	12	2	6
Reputation	2	2	4	4	8
User exp.	1	5	5	3	3
Total			30		27

4.4 Validation of Requirements

In Chapter 2, Principle 6, we emphasized the importance of validating the requirements to avoid unsatisfied stakeholders. Because the requirements form the input for subsequent system development, we must ensure their quality up front to reduce wasted effort downstream, both at the level of the individual requirements and of the work products containing them (Figure 4.14).

We should validate the coverage of stakeholders' needs by our documentation, the degree of agreement among all stakeholders, and the likelihood of our assumptions about the system context before we hand over requirements to the developers or suppliers. Although the level of detail may vary, this applies just as well for iterative as for sequential development approaches.

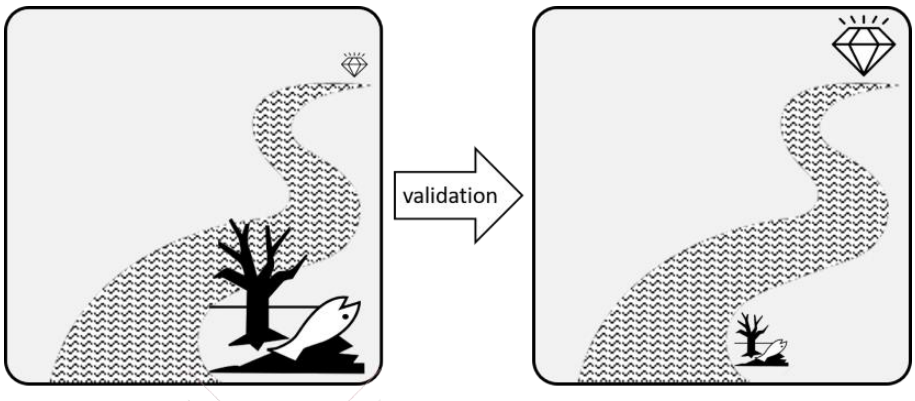


Figure 4.14 Upstream quality reduces downstream waste

Validation adds time and cost to the project, so its efficiency and effectiveness should be a concern of the Requirements Engineer. Therefore, it is important to continuously monitor and analyze defects that occur during development and in operation. If the root cause of such defects appears to be in the requirements, the requirements validation process has somehow failed. Therefore, as a Requirements Engineer, you should continuously and actively look for opportunities to improve it.

4.4.1 Important Aspects for Validation

Regarding the concept of validation, certain aspects are important to get the maximum value from it (see also [PoRu2015]):

- Involving the correct stakeholders

  As a Requirements Engineer, you need to decide who you want to invite to participate in the validation. In this respect, one important aspect that you have to consider is the degree of independence between the people involved in the elicitation of the requirements and those validating them. A low level of independence (inviting stakeholders who have already participated in the elicitation) is cheap and easy to organize but may overlook certain defects because of the own focus, blind spots, conflicting interests, or flawed assumptions of these persons. A higher degree of independence (for instance, by inviting external reviewers or auditors) takes more time and effort to organize and perform and brings higher (initial) costs but may in the long run be more effective in finding more and more severe defects. Consequently, higher risk in the project scope and/or the system context asks for a higher degree of independence.

- Separating the identification and the correction of defects

  It may be tempting to fix every defect as soon as it has been detected. However, this usually proves to be neither an efficient nor an effective way of working, as defects may influence each other. A defect found later during validation might invalidate the fixing of an earlier one. A requirement initially marked as defective might prove to be correct when all requirements have been studied. You might decide not to fix some (minor) defects in view of the effort involved related to the total set of defects found. And after all, people involved in validating requirements should concentrate on finding defects and not on developing ideas on how to fix them. Therefore, the recommendation is to first select (a coherent set of) requirements for validation and to decide whether or not to fix certain defects found only after checking the whole set.

- Validation from different views

  A proper validation is always a group effort, not an activity performed by Requirements Engineers on their own. The best results are achieved when validation is performed by an interdisciplinary team in which selected participants contribute their own expertise. In general, we can say that the input, the output, and peers should be represented. In iterative projects, the current agile team is a reasonable choice,

but the degree of independence may be low and additional validators should be invited; in sequential projects, a specific team may be composed for each separate validation effort. Depending on the phase of the project, input from business, users, developers, testers, operators, and application managers is useful; sometimes, subject matter experts or specialists on topics such as performance, security, and usability can be added.

- Repeated validation

  In sequential projects, most requirements are elicited and documented in the initial phase and validated thoroughly at the end of that phase. However, this should not be the only moment for validation. During the rest of the project, new insights can lead to the original set of requirements being updated, detailed, and expanded. This might threaten the quality, coherence, and consistency of the requirements and thus additional validations may be required. These are often planned at project milestones.
  In iterative projects, many of the agile rituals include validation efforts. Sprint planning, backlog refinement, sprint reviews, and even daily standups offer opportunities to validate and improve the requirements. However, these efforts often focus on individual, detailed requirements and the big picture may be neglected. An initial validation of the complete product backlog at the start of a project or increment is a good beginning. Other useful initiatives are repeated hardening sprints and additional overall validation at release times.

4.4.2 Validation Techniques

As for other techniques, the Requirements Engineer can choose from a large toolbox of validation techniques that differ in formality and effort. Many factors influence the selection of these techniques—for instance, the software development life cycle model, the maturity of the development process, the complexity and risk level of the system, legal or regulatory requirements, and the need for an audit trail.

Often, in the course of a project, the degree of effort and formality increases towards the end, as final decisions about the system and its implementation have to be taken. Also, you will see that the amount, value, and level of detail of feedback from the stakeholders increase as the work products to be validated become more concrete and detailed. This entails the application of different validation techniques in different stages of the project. At the beginning of a project, frequent short, lightweight validation and feedback cycles are preferred, as is usual in agile approaches. This ensures quality right from the start. Later in the project, more formal and time-consuming one-off techniques will prevail.

Apart from that, you can also observe a change in the focus of validation activities. In early phases of a project, the techniques are mostly used to validate the *specification* of requirements. In later phases, the focus of the same techniques may shift to validating their *implementation*.

In general, we discern three categories of validation techniques (see Figure 4.15):

- Review techniques
- Exploratory techniques
- Sample development

Review techniques and sample development are called static, as they concentrate on analyzing the specifications of a system without executing it. In exploratory techniques, the validation focuses on the actual (or simulated) behavior of the system in operation; these techniques are called dynamic.

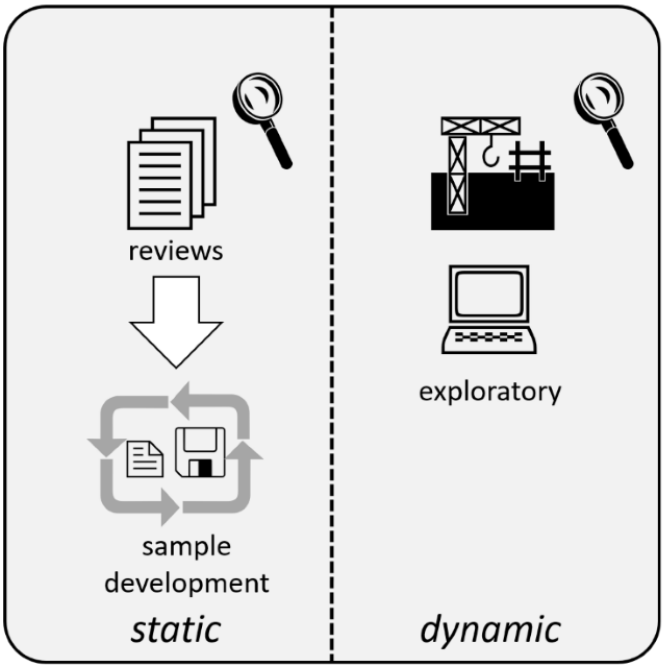


Figure 4.15 Categories of validation techniques

The common feature of *review techniques* is that they rely on visual study of early and intermediate work products. They range from informal to very formal and can be applied from the very beginning of a project until the implementation of the system. In most cases, reviewing the requirements is limited to the earlier phases of a project. Typically, in a review, we check *static* work products that define or describe how the system should work. For more information about reviewing, see [OleA2018].

Informal reviews usually follow the author–reviewer cycle. An author sends a work product to a group of people with the request to validate it. Usually, this is a small group of team members, peers, and/or users involved in the project. Authors may select the group by themselves or its composition may be prescribed by company regulations. After a short (but often not predefined) period, the author collects all review comments and uses them to update the work product at hand. It is good practice to document the comments in a review register and to keep track of the way in which they are processed. However, due to the informal nature of this type of review, authors are free to decide whether and how to use the

comments. Often, the review is repeated over several draft versions until the author is satisfied with the quality.

As they are informal, you might expect little benefit from these kinds of reviews for validating and improving the quality of requirements. However, if all participants are committed to quality, and are able and willing to spend enough time on the review process, informal reviews are an easy, cheap, and approachable means of validation. In fact, this approach is common for early drafts. For the final version of a work product, a more formal technique may be a better choice.

Formal reviews follow a prescribed way of working. They are often used for important or milestone work products, for final versions, and in situations where high risks are at stake. While there are many flavors of formal reviews, they can be divided into two main groups:

- Walkthroughs

  The essence of a *walkthrough* is that the author of a work product explains it step by step to an audience in an interactive session. In practice, walkthroughs come in two variants, where (1) reviewers join the meeting without any preparation and listen to the author, asking ad hoc questions; or (2) they obtain the work product before the meeting and will prepare questions for the author. Participants in the audience can make comments, identify flaws, and suggest alternatives. The author gives more explanation if necessary and can discuss solutions for weaknesses identified and weigh alternatives against the original ideas. There are two occasions where walkthroughs are best applied: (a) in an early project phase to discuss the feasibility of a certain system concept or solution outline; and (b) on the transfer of an intermediate work product to another party who will use it as input for subsequent development. In iterative projects, walkthroughs are mostly present in the form of regular refinement sessions prior to an iteration and sprint reviews at the end of it.

- Inspections

  Inspections are among the most formal review techniques. Here, the responsibility for the review lies not with the author but with an independent review leader, often called *moderator*. An inspection is normally performed in the form of a meeting with the moderator, the author, and a group of *inspectors*.
  The inspectors are selected from peers, business, users, and/or experts. They are asked to check the work product based on their specific expertise, to verify its adherence to applicable standards, norms, and regulations, and to evaluate it against agreed objectives. Often, this check by the inspectors is performed during thorough individual preparation prior to the actual meeting, guided by detailed checklists. In the review meeting, the author participates in the role of a listener, explaining things that are not clear and trying to understand the comments of the inspectors and the consequences for the work product. Typically, an inspection follows a strict and documented process that is managed by the moderator and focuses on finding defects and measuring defined quality aspects and provides a detailed audit trail. In this form, inspections are often used to decide on the release of a work product for a

next step in the development process, or even for final implementation. Inspections are mostly applied in (safety-) critical systems and business processes. In agile approaches, this formal way of reviewing is incorporated in the methodology itself—for example, with the Scrum ceremonies (refinement, planning, sprint review).

Exploratory techniques offer a group of stakeholders and prospective users the opportunity to gain hands-on experience with an intermediate version of (part of) the system under development. In contrast to reviews, exploratory techniques are *dynamic*: they look at the (actual or simulated) behavior of the operative system as experienced by the users through the user interfaces. The participants are invited to use the system in a way that is similar to the intended use in production. They are relatively free to do so but sometimes certain guidance is given. After a period of use, the participants report their experiences and their feedback on the current behavior of the system to the Requirements Engineer. This may include defects found and suggestions for improvement.

Exploratory techniques are common in iterative and design thinking development approaches. In fact, the usual incremental development, starting with the release of a *minimum viable product (MVP),* followed by the addition of more functionality step by step, while carefully measuring market reactions and adjusting the system accordingly, can be seen as an exploratory validation of the requirements in production.

Common exploratory techniques include:

- Prototyping

  In validation with *prototyping*, a specific early version of the system is given to a group of stakeholders for evaluation. This version may be explicitly built for validation purposes, after which it is discarded; we call this an exploratory or throwaway prototype. Of course, evolutionary prototypes, which are continuously updated and extended until they end up in the final product, can also be used for validation during their development. The essence of any prototype is that, from the outside, it looks like the intended system, allowing stakeholders to gain hands-on experience while the internal structure may still be unfinished, inoperative, or even completely missing. When using a prototype for validation, you may have it built to check a specific characteristic, such as user interface, security, or performance.

- Elicitation and validation go together

  As we saw in Section 4.2.3, prototyping and storyboarding can also be used as elicitation techniques. In fact, these techniques support both elicitation and validation, going hand in hand: while validating requirements elicited at an earlier point in time, you will almost certainly detect new requirements in the feedback from the participants. Both aspects of prototyping are very prominent in design thinking approaches (see [LiOg2011]).

- Alpha testing and beta testing

  In alpha testing and beta testing, a fully featured, completely working pre-production version of the system is provided to end users for operation with the intended business processes in a realistic environment.
  Alpha testing is done at the developer's site in a simulated environment. The group of participants is relatively small, some guidance may be given, and it is possible to observe the interaction of the users with the system—for instance, in a usability lab.
  Beta testing is conducted at the end user's sites in real production (or in whatever environment the end users decide). The system is offered (mostly for free) to a (sometimes selected but usually unknown) group of users, with the implicit request to validate its looks and behavior. In beta testing, it is important to stimulate all participants to give their feedback and to provide an easy way to do so. Analyzing this feedback after a prolonged period of use can give valuable clues to the quality of the requirements. It is particularly useful for checking certain assumptions made during elicitation and development.

- A/B testing

  A/B testing is often performed with a released version of the system in the fully operational environment but can also be applied with pre-release versions in a protected test environment. The essence of A/B testing is that the system is offered to different (mostly randomly selected) groups of users in two variants that differ in design or functionality and realize the user goals in a different way. The reaction of both groups is measured and compared; this works best when the groups are large enough to allow for statistical analysis. The analysis will then give information on the quality of the underlying requirements and on the correctness of previous assumptions. A/B testing has a prominent role in *The Lean Startup*, one of the design thinking approaches (see [Ries2011]).

In *sample development*, you provide a set of requirements as input for developers; they try to produce some common intermediate work products (e.g., designs, code, test cases, manuals) based on this input. The system itself is not operative (yet), so this kind of validation is *static*, just like in reviewing. During this effort, the developers may detect flaws such as unclarities, omissions, and inconsistencies that prevent them from producing their intended output. Of course, these flaws will be fixed. At the same time, however, the quantity and severity of the flaws detected is an indication of the quality of the requirements. If this quality is not sufficient, more validation is necessary—for instance, additional reviews.

A similar validation can be performed by Requirements Engineers themselves. In that case, you try to document a set of requirements in a different form of representation to the original type: commonly, converting a requirements specification created in natural language into a relevant model, or a specific model into a textual description. This exercise is especially useful for detecting omissions. If you encounter serious problems in this conversion, this indicates the need for additional validation.

4.5 Further Reading

Glinz and Wieringa [GlWi2007] explain the notion and importance of stakeholders. Alexander [Alex2005] discusses how to classify stakeholders. Bourne [Bour2009] deals with stakeholder management. Lim, Quercia and Finkelstein [LiQF2010] investigate the use of social networks for stakeholder analysis. Humphrey [Hump2017] discusses user personas.

Zowghi and Coulin [ZoCo2005] present an overview of requirements elicitation techniques. Gottesdiener [Gott2002] has written a classic textbook on workshops in RE. Carrizo, Dieste and Juristo [CaDJ2014] investigate the selection of adequate elicitation techniques.

Maalej, Nayebi, Johann and Ruhe [MNJR2016] discuss the use of explicit and implicit user feedback for eliciting requirements. Maiden, Gitzikis and Robertson [MaGR2004] discuss how creativity can foster innovation in RE.

The book by Moore [Moor2014] is a classic about conflict management. Glasl [Glas1999] discusses how to handle conflicts. Grünbacher and Seyff [GrSe2005] discuss how to achieve agreement by negotiating about requirements when validating requirements or resolving conflicts.

Validation is covered in any RE textbook; see [Pohl2010], for example.

5 Process and Working Structure

Whenever work has to be done in a systematic way, a *process* is required to shape and structure the way of working and the creation of work products.

> **Definition 5.1. Process:**
> A set of interrelated activities performed in a given order to process information or materials.

A Requirements Engineering (RE) process organizes how RE tasks are performed using appropriate practices and producing work products required. However, there is no proven, one-size-fits-all RE process (see Section 1.4). Consequently, Requirements Engineers have to configure a tailored RE process that fits the given situation.

The RE process shapes the information flow and the communication model between the participants involved in RE (for example, customers, users, Requirements Engineers, developers, and testers). It also defines the RE work products to be used or produced. A proper RE process provides the framework in which Requirements Engineers elicit, document, validate, and manage requirements.

In this chapter, you will learn about the factors that influence the RE process and how to configure an appropriate process from a set of process facets.

5.1 Influencing Factors

There are a variety of influencing factors to consider when configuring an RE process. Before starting with the configuration of an RE process, these factors need to be investigated and analyzed.

On the one hand, such analysis provides information about how to configure the RE process. For example, when the analysis indicates that stakeholders have only a vague idea about their requirements, an RE process should be chosen that supports the exploration of requirements. On the other hand, the influencing factors constrain the space of possible process configurations. For example, if the stakeholders are available only at the beginning of a system development project, a process that builds upon continuous stakeholder feedback would not be suitable. Below, we discuss important factors for the RE process.

Overall process fit. When defining or configuring an RE process, it is vital to know and understand the overall development process chosen for the system to be developed—defining an RE process that does not fit the overall process does not make sense. The overall process may require work products that the RE process must deliver. The terminology used for the RE process should be aligned to the terminology of the overall process. In particular, the terminology for the work products must be aligned. This helps avoid confusion and misunderstandings. It also makes the introduction of the RE process as well as the training

and coaching of the people who have to work according to the process easier. For example, if the system is developed using a linear, plan-driven process that relies on the existence of a comprehensive system requirements specification and a system glossary at the end of the requirements phase, the RE process chosen must fit into the requirements phase of the overall process and produce the two work products required.

Development context. The development context also informs the RE process. Things to consider include the customer-supplier-user relationship, development type, contract issues, and trust. When analyzing the development context, a couple of questions need to be answered:

- Customer-supplier-user relationship: Is there a designated customer who orders the system and pays for it and a supplier who develops the system? Are customer and supplier part of the same organization or do they belong to different organizations? If the former is the case, which people act in the role of customer and which act as supplier? Who are the users of the system? Do the users belong to the customer's organization?

  If not, do they use the system as a product or service for interacting with the customer (for example, in electronic business) or do they buy the system as a product or service from the customer (for example, a mobile app)?

- Development type: What is the organizational framework for the development of a system? Typical types include:
  - A supplier specifies and develops a system for a specific customer who will use the system.
  - An organization develops a system with the intention to sell it as a product or service to many customers in a certain market segment.
  - A supplier configures a system for a customer from a set of ready-made components.
  - A vendor enhances and evolves an existing product.
- *Contract*: Is there a contract or similar agreement that formally defines deliverables, costs, deadlines, responsibilities, etc.? Contracts may be classic fixed-price contracts between a customer and a supplier, with fixed functionality, deadlines, and cost, or may just give a financial framework, while the functionality is defined iteratively.
- *Trust*: Do the parties involved trust each other? If, for example, the customer and the supplier do not trust each other, the requirements have to be specified in more detail than would be necessary in a trust-based relationship.

Stakeholder availability and capability. The availability of stakeholders constrains the configuration options for the RE process. For example, a process requiring continuous close interaction with stakeholders cannot be chosen if core stakeholders are available only for a short period of time at the beginning of the process.

The capability of the stakeholders also influences the process: the less stakeholders are able to express their needs clearly, and the less they know their actual needs, the more the RE process must accommodate the exploration of requirements.

Shared understanding. Only little Requirements Engineering is needed when there is a high degree of shared understanding (see Chapter 2, Principle 3) between stakeholders, Requirements Engineers, designers, and developers about the problem and the requirements. Consequently, the better the shared understanding, the more lightweight the RE process can be [GlFr2015].

Complexity and criticality. The degree of detail to which requirements need to be specified depends strongly on the complexity and criticality of the system to be developed. When a system is complex and/or critical with respect to safety or security, the RE process chosen must accommodate a detailed specification of the critical requirements, including formal or semi-formal models and strong validation—for example, by verifying models that express prescribed behavior or by building prototypes.

Constraints. Obviously, all influencing factors constrain the space of possible configurations of an RE process. When we talk about constraints, we mean those constraints that are explicitly imposed by, for example, the customer or a regulator. Such constraints may imply the mandatory creation of certain work products and following a mandatory process for producing these work products. Customers or regulators may also demand an RE process that conforms to some given standard.

Time and budget available. If schedules and budgets are tight, the time and budget available for RE need to be used wisely, which typically implies choosing a lightweight RE process. Choosing an iterative RE process helps with prioritizing requirements and implementing the most important ones within the given budget and schedule.

Volatility of requirements. If many requirements are likely to change, it is advisable to choose an iterative, change-friendly RE process.

Experience of Requirements Engineers. The RE process chosen should match the competencies and experience of the Requirements Engineers involved. Otherwise, additional time and budget must be allocated to train and coach the process chosen. It is better to choose a rather simple process that the Requirements Engineers can handle properly than a sophisticated and complicated one that overburdens them.

5.2 Requirements Engineering Process Facets

Defining the RE process from scratch for every RE undertaking is a waste of effort. Whenever the influencing factors allow it, the process should be configured from pre-existing elements. In order to provide guidance on how to configure a proper RE process, we describe three facets with two instances each, together with selection criteria to be considered for each instance [Glin2019]. Later, in Section 5.3, we use these facets to configure RE processes. Figure 5.1 shows an overview of the facets and instances.

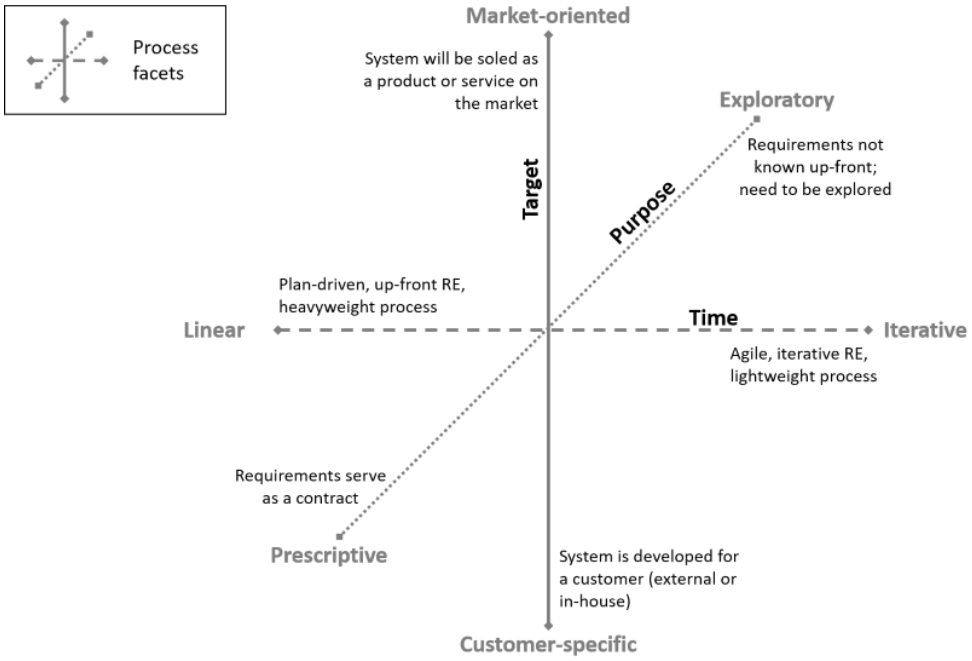


Figure 5.1 RE process facets

The facets can be considered to span a three-dimensional space of process configuration options. Every facet instance comes with criteria for selecting it.

The applicability of these criteria stems from the analysis of the influencing factors discussed in Section 5.1 above. Note that not all criteria need to be fulfilled to choose an instance of a facet.

5.2.1 Time Facet: Linear versus Iterative

The time facet deals with the organization of RE activities on a time scale. We distinguish between linear and iterative processes.

In a *linear RE process*, requirements are specified up front in a single phase of the process. The idea is to produce a comprehensive requirements specification that requires no or only little adaptation or few changes during the design and implementation of the system. Creating a comprehensive requirements specification up front calls for a comprehensive process. Thus, in most cases, linear RE processes are heavyweight processes.

Criteria for choosing a linear RE process:

- The development process for the system is plan-driven and mostly linear.
- The stakeholders are available, know their requirements, and can specify them up front.
- A comprehensive requirements specification is required as a contractual basis for outsourcing or tendering the design and implementation of the system.
- Regulatory authorities require a comprehensive, formally released requirements specification at an early stage of the development.

In an *iterative RE process*, requirements are specified incrementally, starting with general goals and some initial requirements and then adding or modifying requirements in every iteration. The idea is to intertwine the specification of requirements with the design and implementation of the system. Due to short feedback loops and the ability to accommodate change or things forgotten in later iterations, iterative RE processes can be lightweight processes.

Criteria for choosing an iterative RE process:

- The development process for the system is iterative and agile.
- Many requirements are not known up front but will emerge and evolve during the development of the system.
- Stakeholders are available such that short feedback loops can be established as a means of mitigating the risk of developing the wrong system.
- The duration of the development allows for more than just one or two iterations.
- The ability to change requirements easily is important.

5.2.2 Purpose Facet: Prescriptive versus Explorative

The purpose facet deals with the purpose and role of the requirements in the development of a system. We distinguish between prescriptive and explorative RE processes.

In a *prescriptive RE process*, the requirements specification constitutes a contract: all requirements are binding and must be implemented. The idea is to create a requirements specification that can be implemented with no or little further interaction between stakeholders and developers.

Criteria for choosing a prescriptive RE process:

- The customer requires a fixed contract for system development, often with fixed functionality, scope, price, and deadline.
- Functionality and scope take precedence over cost and deadlines.
- The development of the specified system may be tendered or outsourced.

In an *explorative RE process*, only the goals are known a priori, while the concrete requirements have to be elicited. The idea is that requirements are frequently not known a priori but have to be explored.

Criteria for choosing an explorative RE process:

- Stakeholders initially have only a vague idea about their requirements.
- Stakeholders are strongly involved and provide continuous feedback.
- Deadlines and cost take precedence over functionality and scope.
- The customer is satisfied with a framework contract about goals, resources, and the price to be paid for a given period of time or number of iterations.
- It is not clear a priori which requirements shall actually be implemented and in which order they will be implemented.

5.2.3 Target Facet: Customer-Specific versus Market-Oriented

The target facet considers the development type: which kind of development do we target with the RE process? On an elementary level, we distinguish between customer-specific and market-oriented RE processes.

In a *customer-specific RE process*, the system is ordered by a customer and developed by a supplier for this customer. Note that the supplier and the customer may be part of the same organization. The idea is that the RE process reflects the customer-supplier relationship.

Criteria for choosing a customer-specific RE process:

- The system will be used mainly by the organization that has ordered the system and pays for its development.
- The important stakeholders are mainly associated with the customer's organization.
- Individual persons can be identified for the stakeholder roles.
- The customer wants a requirements specification that can serve as a contract.

In a *market-oriented RE process,* the system is developed as a product or service for a market, targeting specific user segments. The idea is that the organization that develops the system also drives the RE process.

Criteria for choosing a market-oriented RE process:

- The developing organization or one of its clients intends to sell the system as a product or service in some market segment.
- Prospective users are not individually identifiable.
- The Requirements Engineers have to design the requirements so that they match the envisaged needs of the targeted users.
- Product owners, marketing people, digital designers, and system architects are primary stakeholders.

5.2.4 Hints and Caveats

It is important to note that the criteria given above are heuristics. They should not be considered as a set fixed rules that always apply. For example, outsourcing the development of the system is done preferably with a prescriptive RE process rather than with an explorative one. This is because the contract between the customer and the supplier is typically based on a comprehensive requirements specification. However, it is also possible to negotiate an outsourcing contract based on an explorative RE process.

There may be prerequisites for choosing certain instances of process facets or the choice may entail consequences that have to be considered. Here are some examples:

- Linear RE processes work only if a sophisticated process for changing requirements is in place.
- Linear RE processes imply long feedback loops: it may take months or even years from writing a requirement until its effects are observed in the implemented system.

To mitigate the risk of developing the wrong system, requirements must be validated intensively when using a linear RE process.

- In a market-oriented process, feedback from potential users is the only means of validating whether the product will actually satisfy the needs of the user segment targeted.
- In an agile setting, an iterative and explorative RE process fits best. Iterations have a fixed length (typically 2–6 weeks). The product owner plays a core role in the RE process, coordinating the stakeholders, organizing the RE work products, and communicating the requirements to the development team.

The three facets mentioned above are not fully independent: the choice made for one facet may influence what can or should be chosen in other ones. Here are some examples:

- Linear and prescriptive are frequently chosen together, which means that when Requirements Engineers decide on a linear RE process, they typically decide on a process that is both linear and prescriptive.
- Explorative RE processes are typically also iterative processes (and vice versa).
- A market-oriented RE process does not combine well with a linear and prescriptive process.

5.2.5 Further Considerations

The degree to which an RE process must be established and followed, as well as the volume of requirements work products to be produced in this process, depends on the degree of shared understanding and also on the criticality of the system.

The better the shared understanding and the lower the criticality, the simpler and more lightweight the RE process can be.

When there is little time and budget available for RE, the resources available must be used carefully. Choosing an iterative and explorative process helps. Furthermore, the process should focus on identifying and dealing with those requirements that are critical for the success of the system.

Finally, the RE process should fit the experience of the Requirements Engineers. The lower their skills and experience, the simpler the RE process should be made—it does not make sense to define a sophisticated process when the people involved cannot enact this process properly.

5.3 Configuring a Requirements Engineering Process

In a concrete system development context, Requirements Engineers or the person(s) responsible for RE have to choose the RE process to be applied. We recommend analyzing the influencing factors (see Section 5.1) first and then selecting a suitable combination of the process facets described in Section 5.2.

5.3.1 Typical Combinations of Facets

Three combinations of facets (or variants thereof) frequently occur in practice [Glin2019]. In the following, we briefly describe each of them and characterize them in terms of their main application case, typical work products, and typical information flow. Furthermore, we provide an example. Figure 5.2 shows the three typical process configurations in the space of the three facets.

Participatory RE Process: Iterative & Explorative & Customer-Specific

A participatory RE process is typically chosen in agile settings when there is a customer who orders a system and a development team that designs and implements it. The focus is on exploring the requirements in a series of iterations in close collaboration between the stakeholders on the customer side, the Requirements Engineers, and the development team.

Main application case:	Supplier and customer collaborate closely; stakeholders are strongly involved in both the RE and the development processes.
Typical work products:	Product backlog with user stories and/or task descriptions, vision, prototypes
Typical information flow:	Continuous interaction between stakeholders, product owners, Requirements Engineers, and developers

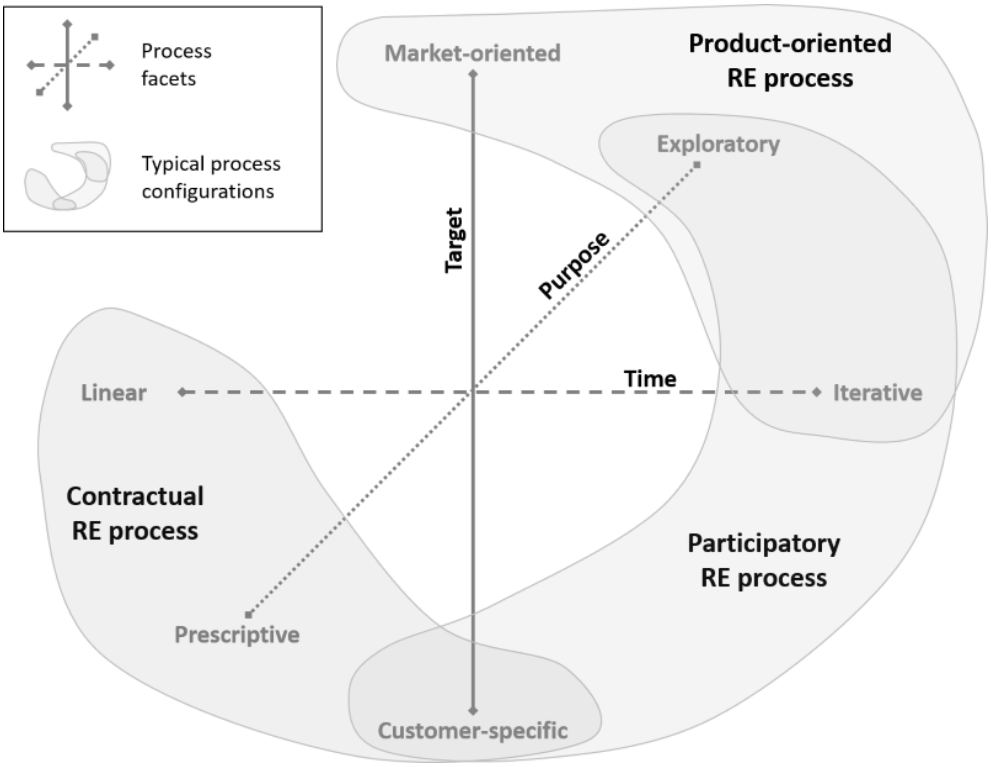


Figure 5.2 Three typical RE process configurations and their relationship to the three facets

Example: In an insurance company, the business unit that sells corporate insurances to small and medium-sized enterprises has an idea about a new product for insuring customers against the damage incurred by a hacker attack. They contract the corporate IT unit of the company to form a development team with the task of designing and developing a new application that can handle the new insurance product within the existing insurance sales support system. Also, the existing insurance contract management system needs to be adapted accordingly. Beyond some initial requirements, the contracting business unit has no clear idea how the new product should look and how it should be supported by the corporate IT systems. Corporate IT adopted agile development for all their projects some years ago.

In this situation, a participatory RE process is appropriate. It fits the overall agile process that corporate IT will employ to develop the new system and adapt the existing ones. Stakeholders from the business unit and Requirements Engineers from corporate IT can jointly elicit the requirements for the new insurance product. As the process is iterative, the development team can develop a prototypical minimum marketable product (MMP) that helps the management of the business unit to decide whether or not to include the product envisaged in their portfolio or discard the idea. There is a clear customer-supplier relationship between the business unit and corporate IT, so a customer-oriented RE process fits.

Contractual RE Process: Typically Linear & Prescriptive & Customer-Specific

A contractual RE process is typically chosen when the development of a system is tendered and outsourced to a provider with a contract based on a comprehensive requirements specification. It is also a suitable process for RE in large system development projects that apply a waterfall-style development process.

Main application case:	The requirements specification constitutes the contractual basis for the development of a system by people not involved in the specification and with little stakeholder interaction after the requirements phase.
Typical work products:	Classic system requirements specification, consisting of textual requirements and models
Typical information flow:	Primarily from stakeholders to Requirements Engineers

Example: A car manufacturer is developing a new car platform, from which a family of car models will be derived. A major design decision for the new platform is to get rid of the dozens of electronic control units (ECUs) currently used in the cars and replace them with a single control computer that runs a stack of driving control and driving assistance applications. The goal is to save hardware costs, get rid of unwanted interactions between ECUs, and reduce both time and effort for performing updates of the software. Engineers who are responsible for the electronic systems of the new platform have written a customer requirements specification. The company has contracted a large manufacturer of automotive control systems to create a system requirements specification for the new centralized car control system. Later, the car manufacturer will tender the design and implementation of the system based on that specification. The manufacturer will require the implementation to be performed in several iterations in order to ease testing and integration of the system with the new car platform.

In this situation, a contractual RE process is appropriate. The overall process is linear: the system will be designed and implemented only after the requirements specification has been completed. The fact that the implementation will be iterative does not impact the RE process. Depending on the quality of the existing customer requirements specification and the availability of the stakeholders at the car manufacturer, a linear or an iterative RE process should be chosen.

Obviously, a customer-oriented RE process is needed. The existence of a customer requirements specification and the fact that the system requirements specification will be used to tender the design and implementation of the system call for a prescriptive RE process.

Product-Oriented RE Process: Iterative & Explorative & Market-Oriented

A product-oriented RE process is typically chosen when an organization is developing a system as a product or service for the market. In most cases, a product-oriented RE process comes together with an agile product development process. The product owner and digital designers play major roles in this process: they strongly influence and shape the product.

Main application case:	An organization specifies and develops software in order to sell or distribute it as a product or service
Typical work products:	Product backlog with user stories and/or task descriptions, vision, prototypes, user feedback
Typical information flow:	Interaction between product owner, marketing, Requirements Engineers, digital designers, and developers plus feedback from customers/users

Example: A media company tasks its internal IT with a total renewal of the mobile news app that the company sells to subscribers (with some content being freely accessible). From user feedback, the company maintains a long log of customer criticism and improvement suggestions.

In particular, many users criticize the existing app for not being responsive enough, for having bad support for reporting problems and suggestions, and for not supporting two-finger zooming of text or images. The marketing department of the company also perceives the layout of the app to be outdated. They predict that with a fresh layout, more subscribers could be gained. The CEO of the company has decided that the IT department shall collaborate with an external design agency for the visual appearance of the app. The management of the media company wants a minimal product version as a proof of concept, and then new intermediate versions every three weeks that can be reviewed by the marketing department and the company's board of executives.

In this situation, a product-oriented RE process fits best. Although there is a customer-supplier relationship between the management of the company and its internal IT department, the focus is clearly on creating a renewed product in the segment of mobile news applications. The RE process needs to be explorative, as the requirements beyond the information in the existing log of user feedback are not clear. The overall development process has to be iterative according to the decision of the management of the company. As the requirements need to be explored, an iterative RE process is the best fit here.

5.3.2 Other RE Processes

The three combinations described above cover many of the situations that occur in practice. However, there may be situations where none of the aforementioned process configurations fit. For example, regulatory constraints may impose the use of a process that conforms to a given standard, such as ISO/IEC/IEEE 29148 [ISO29148]. In such a case, the RE process has to be created by process experts from scratch or one of the aforementioned configurations has to be tailored so that it is adapted to the given situation.

5.3.3 How to Configure RE Processes

We recommend a five-step procedure for configuring an RE process.

1. *Analyze the influencing factors.* Analyze your situation with respect to the list of influencing factors from Section 5.1.
2. *Assess the facet criteria.* Based on the analysis from step 1, go through the list of facet selection criteria given in Section 5.2. You may assign each criterion a value on a five-point scale (––, –, 0, +, ++).
3. *Configure.* If the criteria analysis yields a clear result with respect to the three typical configurations mentioned above, choose that configuration. Otherwise, choose a different process tailoring, guided by the general goal of mitigating the risk of developing the wrong system. For example, imagine a situation where the customer demands a system requirements specification to be created up front, which calls for a linear, prescriptive RE process. However, in your first meetings with the customer, you have noticed that for an important subsystem, the customer has no clear idea what to build, which calls for an explorative RE process. A potential solution could be to choose a contractual RE process as the general RE process framework but create a subproject that stepwise elicits the requirements for that important subsystem,

creating prototypes in two or three iterations (guided by a participatory RE subprocess), and then feed the results into the system requirements specification.

4. *Determine work products.* Based on your analysis and process configuration, define the main RE work products that will be produced. Make sure that the RE work products are aligned with the work products of the overall development process.
5. *Select appropriate practices.* For the tasks to be performed—for example, elicitation of requirements—select the practices that fit best in the given situation. Many of these practices, including hints about where and when to apply them, are presented in Chapters 2, 4, and 6 of this handbook.

There is no proven, one-size-fits-all RE process. Based on an analysis of influencing factors, a specific RE process needs to be tailored for every RE undertaking. A simple way of tailoring is configuring an RE process from a set of process facets.

5.4 Further Reading

Armour [Armo2004] and Reinertsen [Rein1997], [Rein2009] provide general thoughts on processes and information flows in processes.

Although the textbook of Robertson and Robertson [RoRo2012] is entitled "Mastering the Requirements Process," this is a general textbook on all aspects of RE.

Wiegers and Beatty [WiBe2013] provide a chapter about improving RE processes. The book by Sommerville and Sawyer [SoSa1998] contains a collection of good practices to be used in the framework of RE processes.

6 Management Practices for Requirements

Requirements are not carved in stone, eternally present from past to future; they are alive! They are born through elicitation, grow up through documentation, and are shaped through validation. As adults, they go to work through implementation and after a—hopefully—long and prosperous life in operation, they retire in oblivion. Throughout their life cycle, their parents, the Requirements Engineers, take care of them. We nurse them in their infancy, teach them in their youth, escort them in their relationships, and help them find a good job in a healthy system. That is what we call requirements management.

Of course, there are better, more formal, definitions of requirements management. The ISO/IEC/IEEE 29148:2018 [ISO29148] standard defines requirements management as "*activities that identify, document, maintain, communicate, trace and track requirements throughout the life cycle of a system, product or service.*". In the CPRE glossary [Glin2020], requirements management is defined as "*The process of managing existing requirements and requirements related work products, including the storing, changing and tracing of requirements.*". The CPRE glossary also tells us that *requirements management* is an integral part of Requirements Engineering: "*The systematic and disciplined approach to the specification and management of requirements with the goal of …*".

Requirements management can occur at different levels:

- The individual requirements
- The work products that contain these requirements
- The system related to the work products and the requirements contained therein

In practice, requirements management is primarily performed at the work product level. Usually a work product contains several individual requirements (e.g. an external interface description), while other work products contain only a single requirement (e.g. a single user story in an agile project) or they represent the whole set of requirements for a system (e.g. software requirements specification). Be aware that all work products of all three levels must be managed, and make sure that you know the relationships between them.

The text above outlines the *what* of requirements management. The rest of this chapter is devoted to the *how*: all kinds of practices that are applicable to make requirements management work. Before we dive into the details of requirements management, let us consider some leading principles for making it work. If you want to manage something, you must be able to recognize it, to store it, and to find it again. Therefore, unique identification, an appropriate degree of standardization, avoidance of redundancy, a central repository, and managed access are a must.

In Section 6.1, we take a short look at situations that influence the value, importance, and effort involved in requirements management.

Section 6.2 follows the requirements in their life cycle as part of work products that Requirements Engineers and other IT staff produce and use while developing, implementing, and operating an IT system.

During the lifecycle of a requirement, multiple versions of work products (and the requirements they contain) are created, starting with an early 0.1 draft that, after a series of major and minor changes, evolves into, say, a 3.2 final version. Version control is discussed in Section 6.3.

When developing and using IT systems, it is impractical to deal with all requirements on an individual basis. Therefore, coherent sets of requirements are recognized as configurations and baselines, as explained in Section 0.

In order to handle work products and requirements efficiently, we must be able to identify them and collect data about them. That is the topic of Section 6.5.

Section 6.6 looks at requirements traceability. Traceability is an especially important quality characteristic of requirements, as you may have already understood when reading the definitions of requirements management above. Without traceability, it is impossible to link the actual behavior of a system to the original demands of the stakeholders.

Section 6.7 deals with the changes to requirements that occur during their lifetime. In the first phases of their existence, changes can be frequent, but after validation, requirements should be stable. However, changes will still occur. To apply them in an orderly manner, a defined process for handling change should be in place.

By nature, requirements differ in importance and value. Usually, resources to elaborate them are limited, so not every requirement will make it to implementation. This means that stakeholders will have to decide when a certain requirement will be implemented or even whether or not it will be implemented at all. Prioritization, described in Section 6.8, can underpin this decision.

6.1 What is Requirements Management?

In the introduction, we have already seen that requirements management means the management of existing requirements and requirements-related work products, including storing, changing, and tracing the requirements. But why manage them at all?

We manage requirements because they are living things; they are created, used, updated, and deleted again during both their development and operation. And during this whole life cycle, we must make sure that all parties involved have access to the correct versions of all requirements that are relevant to them. If we do not manage requirements properly, we face the risk that some parties may overlook requirements, stick to outdated requirements, work with wrong versions, overlook relationships, and so on. This can seriously hinder the efficiency and effectiveness of system development and usage. In other words: the value of proper requirements management lies in the improved efficiency and effectiveness of a system.

This means that the value of requirements management cannot be separated from the value of the system in question and its context. In practice, we can see huge differences in the importance and level of requirements management and the effort involved [Rupp2014], ranging from an informal subsidiary task of a Requirements Engineer with a spreadsheet, to

a full-time function of a dedicated requirements manager with a tool-supported database of requirements.

More thorough requirements management is needed with larger numbers of requirements, stakeholders, and developers, with a longer expected lifetime, more changes or higher quality demands on the system, and with a more complex development process, more strict standards, norms, and regulations, including the need for a detailed audit trail.

Often, we see that requirements management is somewhat neglected at the beginning of a project, when a small team is working on an obvious set of high-level requirements. Later on, complexity increases and the team loses the overview, resulting in quality problems and reduced efficiency. Then, a lot of effort has to be spent on catching up with the required level of control. It is more efficient to invest some effort right from the start of a project to set up the requirements management resources and processes with the expected demands at the end in mind.

6.2 Life Cycle Management

As stated in the introduction, requirements and work products that contain requirements have a life. We see them being created, elaborated, validated, consolidated, implemented, used, changed, maintained, reworked, refactored, retired, archived, and/or deleted. That is what we mean by their life cycle: during its life, a requirement can be in a limited number of states and can show a limited number of state transitions based on explicit events in the context. Figure 6.1 shows a simplified *statechart* as a model for the life cycle of a single requirement (overview only, state transitions are not shown; for instance, the transition from the composite state *Under development* to *In production* may be triggered by a go-live decision from the product owner).

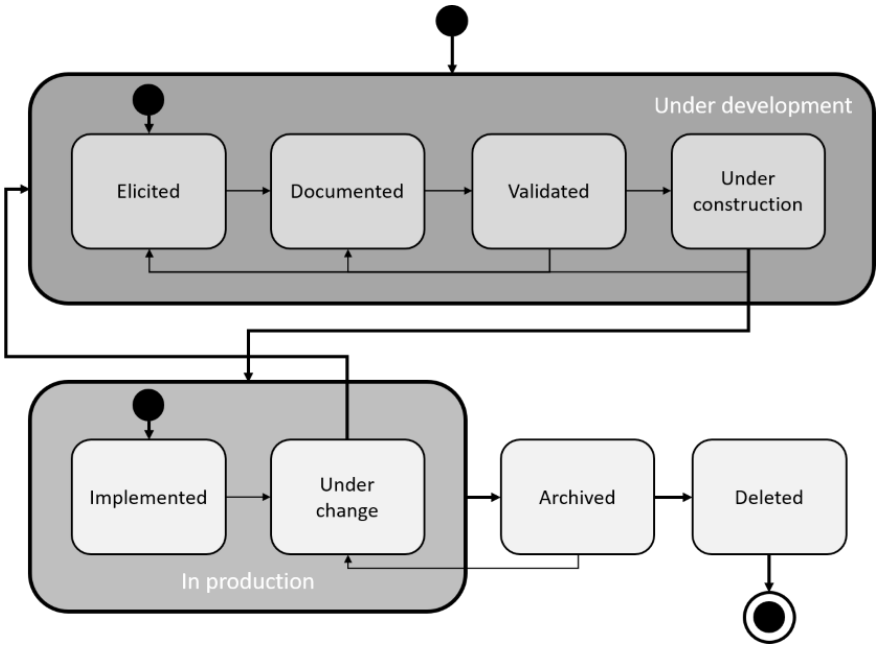


Figure 6.1 Simplified statechart of a requirements life cycle

A complicating factor is that work products and individual requirements have their own different life cycles that only partially overlap. As an example, think of a work product *definition study* in the state *under change*; this does not necessarily mean that all requirements contained in the work product have to be changed. And for the same *definition study*, the state *implemented* makes no sense; only some requirements in it will be implemented—or better: certain code, based on these requirements.

Another complicating factor may be that in practice, the view of the life cycle of requirements is different for different roles. For you as a Requirements Engineer, to trace your work you are interested in different states to the project manager, and other states again compared to the product manager or a change manager: in the diagram above, your interest might end at *validated*, while for the project manager, it only starts at *documented*.

Requirements Engineers actively manage the life cycle of their work products. Life cycle management implies:

- Defining life cycle models for your work products and the requirements contained in them with
- The states that a work product or requirement can take
- The transitions allowed between these states
- The events that trigger the transition from one state to another
- Ensuring that only explicitly allowed transitions occur
- Recording the actual states that the work products and requirements take
- Recording the actual transitions that occur
- Reporting on these states and transitions

In simple words: make sure that you know the state that your requirements were in, are in, and will take, how they can change, and why this all happens.

For instance, as a Requirements Engineer, you could be asked to report who approved which version of a requirement to be released as input for the coding phase. Keeping track of requirements states in their life cycle can also be useful for building dashboards and reporting on the progress of a project. It can be a good way to organize work and identify which requirements to work on first.

The state of a work product under life cycle management is often recorded in an attribute (see Section 6.5). It may also be useful to document the beginning and the end date of that state in attributes. In agile projects, the state of a work product (item) can be derived from its position in the product backlog, task backlog, and/or on the task board. Also, meeting the criteria of the *definition of ready* and the *definition of done* can give relevant information, as meeting these criteria actually means attaining a next state.

The thoroughness and level of detail of the life cycle management should be tailored to the needs of the customer, the project, and the system. For instance, the states under development, in production, and archived might be sufficient. In complex or critical projects, you may need a far more detailed model of the states, strict procedures about state transitions, and an audit trail that shows what happened during the project.

6.3 Version Control

It is common for both, work products and individual requirements as part of a work product, to undergo certain changes during their life cycle (see Section 6.7 for more information on handling these changes). After every change, the work product is different to what it was before: it has become a new version.

We want to control the versions of these work products for two reasons:

- Sometimes changes go wrong. After a while, defects are found, or the intended benefits are not realized. In such a case, we may implement new changes in a next version but we can also decide to go back to a previous version and continue from there. Or maybe, on second thought, we just prefer the earlier version after all.
- We want to know the history of the work product, understand its evolution right from its origin up to its present situation. This may help us when we have to decide on future changes, or just answer questions on why the current work product is what it is.

Version control requires three measures to be in place:

- An *identification* of each version, to distinguish between the different versions of a work product. This is the version number, often supplemented with a version date.
- A clear description of each *change*. You must be able to tell—and understand—the difference between a certain version and its predecessor. This change description must be clearly linked to the version number.
- A strict policy on the *storage* of versions, enabling you to locate and retrieve old versions. Unless storage limitations dictate otherwise, you should preserve all previous versions of all your work products, otherwise you may not be able to restore a version if you need it. On the other hand, unlimited storage will rarely be the case, so it is wise to also have a policy for archiving and cleaning up work products that are no longer used.

Usually, a work product contains multiple requirements. If a single requirement in that work product changes, both that requirement and the work product should get a new version number, while the unchanged requirements in that work product keep their old version number. This might soon become very confusing. A practical solution may be to do version numbering at the work product level only and let all requirements in it inherit the version number and the change history of the work product.

Version numbers are typically composed of (at least) two parts:

- *Version*. In principle, the version starts at *zero* as long as the work product is under development. When it is formally approved, released, and/or launched, we assign it version *one*. After that, the version is increased only for major, substantive updates.
- *Increment*. This mostly starts at *one* and is incremented with every (externally visible) change, on the content side or often only textual or editorial. An additional sub-increment may be used for correction of typos only. The increment *nine* is sometimes used to denote a final version just before approval or release.

A new version number is assigned with each formal change.

Often, a change in the life cycle state of a work product is not considered a reason for incrementing the version number, unless it is accompanied by a change in content or text. If, for instance, a requirement receives the state validated and the version number 1.0 after approval, there is no need to change this version number if the state changes to under construction and subsequently to implemented. The state can finally end in archived but still keep the same version number 1.0.

6.4 Configurations and Baselines

Suppose you preserve, as advised above, all versions of all requirements that you develop during a project. You will then have an ever-expanding database filled with requirements and you will start to lose the overview. One day, your client comes to your desk and asks: "We have implemented your system at all our branches. Now there seems to be a problem with the calculations in our Barcelona office. Can you tell me what version of the calculation requirements they use there?" If you cannot answer that question, you will wish that you had paid more attention to configuration management.

So, what is a configuration? You will find a definition in the CPRE glossary [Glin2020] but in short, for a Requirements Engineer, a configuration is a consistent set of logically related work products that contain requirements. We select this set with a specific purpose, usually to make clear which requirements are or were valid in a certain situation.

This sets the following properties for a correct configuration:

- *Logically connected*. The set of requirements in the configuration belongs together in view of a certain goal.
- *Consistent*. The set of requirements has no internal conflicts and can be integrated in a system.
- *Unique*. Both the configuration itself and its constituent requirements are clearly and uniquely identified.
- *Unchangeable*. The configuration is composed of selected requirements, each with a specific version that will never be changed in this configuration.

- *Basis for reset*. The configuration allows fallback to a previous configuration if any undesired changes appear to have occurred.

A configuration is documented as a work product, with a unique identification, a state, and a version number and date, just like any other work product. However, because a configuration is by definition unchangeable, it will always have only one version (e.g., 1.0).

A configuration always has two dimensions [CoWe1998]:

- The *product* dimension.

  This indicates which requirements are included in this specific configuration. Sometimes, a configuration will contain all available requirements but usually, it is a certain selection—for instance, all requirements that are implemented in the French release of a system. The British release of the same system might then have a different configuration.

- The *version* dimension.

  In a specific configuration, every selected requirement is present in exactly one, and only one, version. It might be the latest version or an earlier one, depending on the purpose of the configuration itself. As soon as even a single different version of a single requirement is selected, this is a new configuration. Imagine a system for which a new release will be implemented with some requirements in a higher version: this new release will then have a different configuration.

Figure 6.2 gives another example of different configurations consisting of specific sets of versions of requirements.

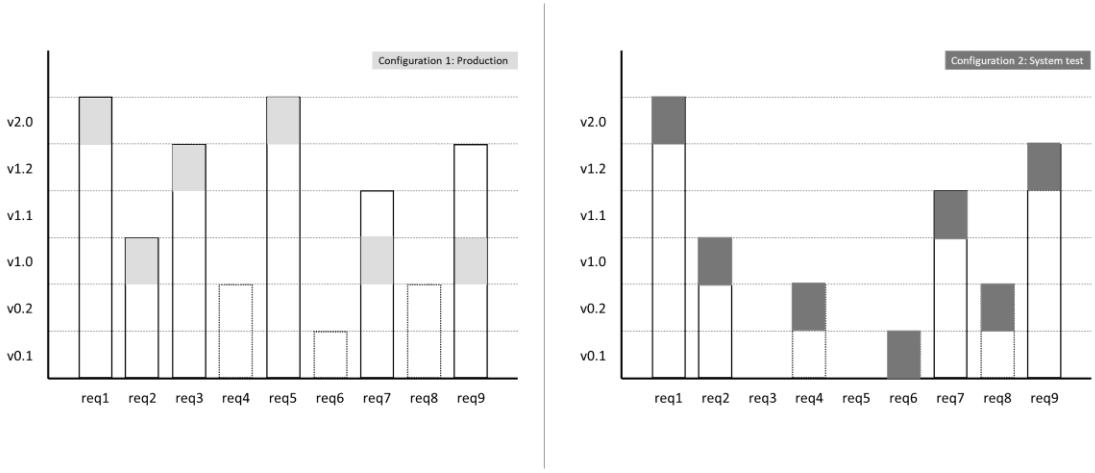


Figure 6.2 Example of configurations

The figure above shows an example of different configurations of a certain system. It shows a collection of nine requirements. Some of them are still in the early stages of development—e.g., requirement 6 with version v0.1. Other requirements have had more versions—for instance, requirement 1, which is finalized and has already had a major update, so is now version v2.0.
The left-hand picture shows the configuration that is currently in production. It consists of R1 v2.0, R2 v1.0, R3 v1.2 (this requirement had two minor updates after implementation), R5 v2.0, R7 v1.0, and R9 v1.0. R4, R6, and R8, being under development, are not present in this configuration, nor are the new versions of R7 and R9.
The right-hand picture shows the configuration that, at the same time, is present in the system test environment. Some requirements (R1, R2) are the same, some are no longer present (R3, R5), the requirements under development (R4, R6, and R8) are included here, and two requirements (R7 and R9) are present in a higher version than in the configuration of the production environment.

In many projects some configurations are treated in a special way: these configurations are called baselines. A *baseline* is a stable, validated, and change-controlled configuration that marks a milestone or another kind of *resting point* in the project. An example can be the configuration at the end of the design phase, just before starting the coding phase, or the configuration that is valid at the go-live of a certain release.

The sprint backlog in an agile project serves as the baseline at the start of the next iteration. Baselines are useful for planning purposes as they represent a stable starting point for a next phase. They are often frozen and set aside as an anchor in the hectic life of a project. If something goes terribly wrong in the project, the team can perform a roll-back to the situation of the baseline and restart from there.

For the Requirements Engineer, it is mainly the configuration of work products containing requirements that is important. But in practice, the configuration within a project has a much broader scope, containing selected versions of the work products of all team members, such as requirements, designs, code and test cases. In complex projects, configuration management can be a full-time job, performed with dedicated tooling.

6.5 Attributes and Views

As a Requirements Engineer, your output consists of all kinds of work products containing requirements. These requirements will have to be managed, otherwise you and your team will quickly lose the overview. To manage the requirements, you have to collect and maintain data about them—metadata, data about data. Metadata makes work products tangible, manageable; through metadata, you can provide and obtain information about the requirements and answer questions that are relevant during and after the project or product life cycle. Think of questions like "*Which requirements are planned for the next release?*" or "*How much effort is this release likely to take?*" or "*How many requirements have a high priority?*"

When considering the requirements as entities about which information is required, the characteristics of these requirements are called *attributes*. In this chapter, we have already seen some common attributes, such as the unique identification, version number, state,

several dates. The attributes to be defined for the requirements depend on the information needs of the stakeholders of the project and the system. At the start of a project, an *attribute schema* should be set that enables the Requirements Engineer to fulfill these needs.

A good starting point can be found in relevant standards. The ISO standard [ISO29148] mentions:

- *Identification*. Each requirement should have a unique, immutable identifier, such as a number, name, mnemonic. Without a proper identification, requirements management is impossible.
- *Stakeholder priority*. The (agreed) priority of the requirement from the viewpoint of the stakeholders. See Section 6.8 for information on how to determine this priority.
- *Dependency*. Sometimes, there is a dependency between requirements. This may mean that a low-priority requirement should be implemented first because another, high-priority requirement depends on it.
- *Risk*. This is about the potential that the implementation of the requirement will lead to problems, such as damage, extra costs, delays, legal claims. By nature, this is an estimate, to be based on consensus among stakeholders.
- *Source*. What is the origin of the requirement, where did it come from? You may need this information for validation, conflict resolution, modification, or deletion.
- *Rationale*. The rationale gives you the reason why the requirement is needed, the objectives of the stakeholders that should be fulfilled by it.
- *Difficulty*. This is an estimate of the effort needed to implement the requirement. It is needed for project planning and estimation.
- *Type*. This attribute indicates whether the requirement is a functional or a quality requirement or a constraint.

There are many ways to store this information. It may be contained in documents or stored in a spreadsheet or database, with the requirements as rows and their attributes as columns. In agile settings, requirements may be recorded on story cards, where the rubrics on the card are the attributes. As discussed in Chapter 7, requirements management tools should offer functionality for storing data about requirements and also reporting on them.

Attributes allow you to provide information about your work products and the requirements contained therein. The simplest way to do so is to produce a report with all the data on all the versions of all requirements. For anything but the simplest system, such a report will be useless as nobody will be able to oversee all the information because it is overwhelmingly complex. Therefore, you should adjust your reports based on the information needs of your target audiences. This is done by using *views* [Glin2020].

A view is an (often predefined) way to filter and sort the data on your work products, resulting in a report that shows precisely what the audience needs, no more, no less. A view is defined with the explicit purpose of delivering relevant information for a specific target group.

We discern three types of views:

- *Selective views*. These views give information on a deliberate selection of the requirements instead of all requirements. For example, a view on only the latest versions of the requirements, or all requirements with the state *validated*, or on the requirements with stakeholder priority *high*; the focus might be on a subsystem, or on the contrary to provide an abstract overview of the system through its high-level requirements only.
- *Projective views*. A projective view shows a selection from all data (attributes) of the requirements—for example, only the identification, the version number, and the name.
- *Aggregating views*. In an aggregating view, you will find summaries, totals, or averages, calculated from a set of requirements. An example would be the total number of requirements per department: e.g., 4 from Sales, 5 from Logistics.

Figure 6.3 gives an example of these types of views.

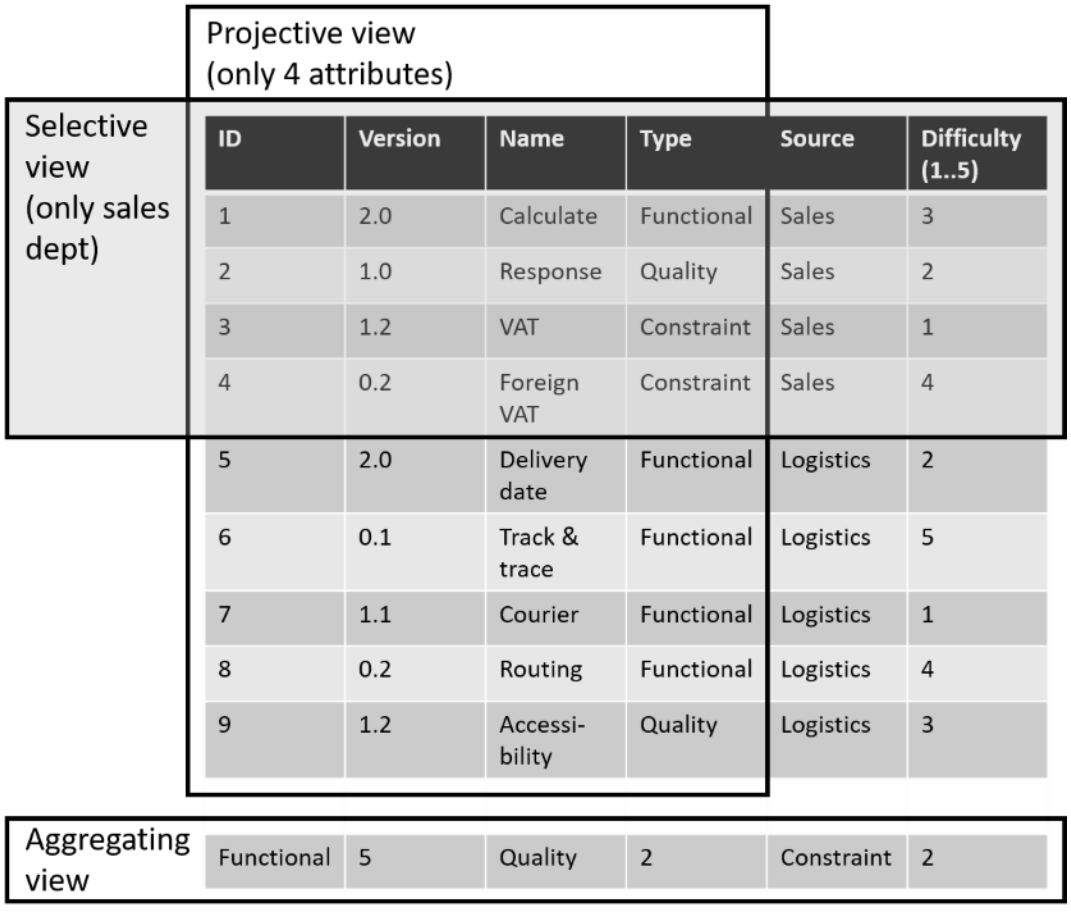


Figure 6.3 Different types of views

In most cases, a combination of views is used—for instance, if you want to provide a list with the IDs, version numbers, names, and types (= projective) of all the requirements for the Sales department (= selective).

6.6 Traceability

Throughout this handbook, we have mentioned the topic of traceability [GoFi1994]. Without proper traceability, Requirements Engineering is hardly feasible, as you cannot do the following:

- Provide evidence that a certain requirement is satisfied
- Prove that a requirement has been implemented and by what means
- Show product compliance with applicable laws and standards
- Look for missing work products (e.g., find out whether test cases exist for all requirements)
- Analyze the effects of a change to requirements (see Section 6.7)

In many cases, especially for safety-critical systems, process standards even explicitly demand the implementation of traceability.

There are three types of questions that can be answered with the aid of traceability (see also Figure 6.4):

- Backward traceability: What was the origin of a certain requirement? Where was it found? Which sources (stakeholders, documents, other systems) were analyzed during elicitation?

  Backward traceability is as well-known as pre-requirements specification traceability.

- *Forward traceability*: Where is this requirement used? Which deliverables (coded modules, test cases, procedures, manuals) are based on it?

  Forward traceability is as well-known as post-requirements specification traceability.

- *Traceability between requirements*: Do other requirements depend on this requirement or vice versa (e.g., quality requirements related to a functional requirement)? Is the requirement a refinement of a higher-level requirement (e.g., an epic refined in a number of user stories, a user story detailed with a number of acceptance criteria)? How are they related?

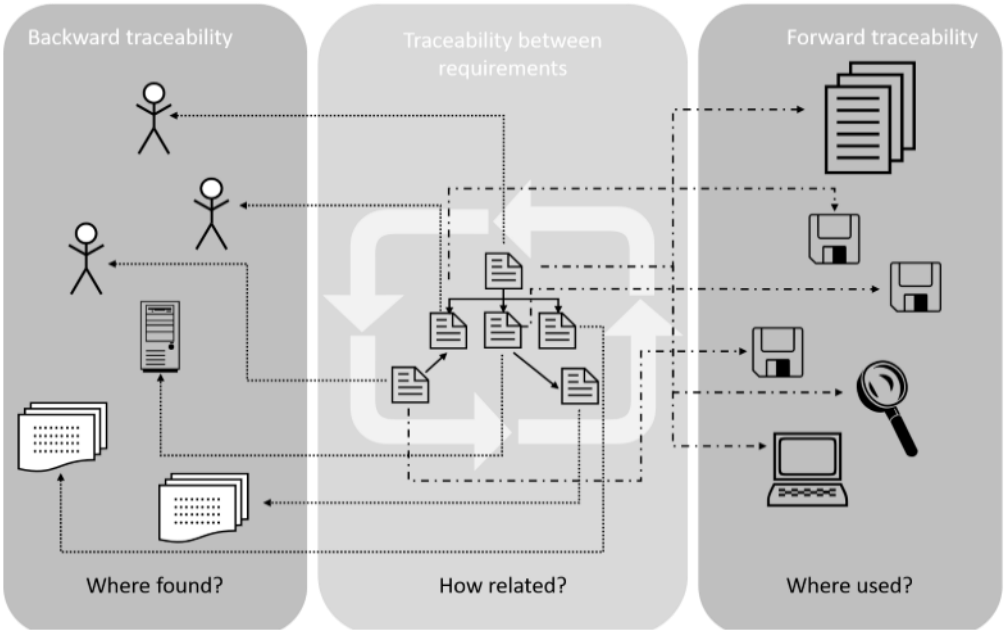


Figure 6.4 Traceability types

There are several ways of documenting traceability. Often, this is done *implicitly*—for instance, by applying document structures, standard templates, or naming conventions. If you identify all your requirements with the code *Req-xxx-nnn*, where *xxx* stands for the department that requested the requirement, everybody will understand that Req-sal-012 is a requirement for the Sales department (for backward traceability). If you publish a document listing all the requirements that will be implemented in the release of July 1st, you are providing implicit forward traceability information.

And if you write a document with a dedicated section on, e.g., price calculations, that could be an example of traceability between requirements. Another example could be a high-level model and a textual description of detailed requirements related to it.

In more complex projects, traceability should (also) be documented *explicitly*. For explicit traceability, you document the relationship between work products based on their unique identification. This can be done in various forms [HuJD2011]:

- Making use of specific attributes such as *Source* suggested by the ISO standard [ISO29148]
- In documents, adding references to predecessor documents, other work products, or individual requirements
- Developing a traceability matrix in a spreadsheet, or a database table (see an example in Table 6.1 below)
- In textual documentation, using Wiki-style hyperlinks
- Visualizing traceability relationships in a *trace graph* (Figure 6.4 is a simplified form of such a graph)

In many cases, a requirements management or configuration management tool (see Chapter 7) provides functionality to support traceability. Managing traceability in a

substantial project can be complicated, especially if you also have to take versioning into account. In such a case, good tooling is indispensable.

Table 6.1 Example of a traceability matrix

Source	R1	R2	R3	R4	R5	R6	R7
Interview Mrs. Smith 06/08	X	X			X		
Summary questionnaire May 12	X			X		X	X
Field observation report 07/03			X	X	X		
Company regulations version 17.a.02			X			X	X
Documentation API HRM system v3.0.2.a	X			X	X		

6.7 Handling Change

"Principle 7: Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage." [BeeA2001]. The founding fathers of the agile movement were crystal clear on this: requirement changes will always occur, whether you like them or not. Many people do not like changes at all, because every change is a risk, a threat to the stability of the project and the system.

However, changing a requirement is not a stand-alone event: it is triggered by changes in the system context, by new insights of the stakeholders, by behavior of competitors, and so on; a law becomes effective, adding a new constraint to the system; due to growing market demand, the performance of the system has to be improved; a competitor system is launched with some *delighter* features that your client wants too. A change should thus be seen as a chance to get a better system, to provide more value to the users.

However, regardless of the situation, every change is also a risk. It can introduce defects, leading to system failure. It can delay the progress of the project. It can take more effort and money than was calculated before. The users may not like it and refuse to work with it. In short, things can go wrong and disturb a previously stable project or system. But that does not mean that changes are bad and should be avoided; it does mean that all changes must be handled carefully to get optimal value at acceptable costs with minimal risk.

In the literature on IT service management (see [Axelos2019]), *change enablement* is described as one of the core practices. This practice ensures that changes are implemented effectively, safely, and in a timely manner in order to meet stakeholders' expectations. The practice balances effectiveness, throughput, compliance, and risk control. It focuses on three aspects:

- Ensuring that all risks have been accurately assessed
- Authorizing changes to proceed
- Managing the change implementation

Change enablement implies that an organization assigns a change authority to decide on the changes and defines a process for handling them. See Figure 6.5 for an outline of this process. These measures are usually tuned to the development approach and the point in time where a change occurs.

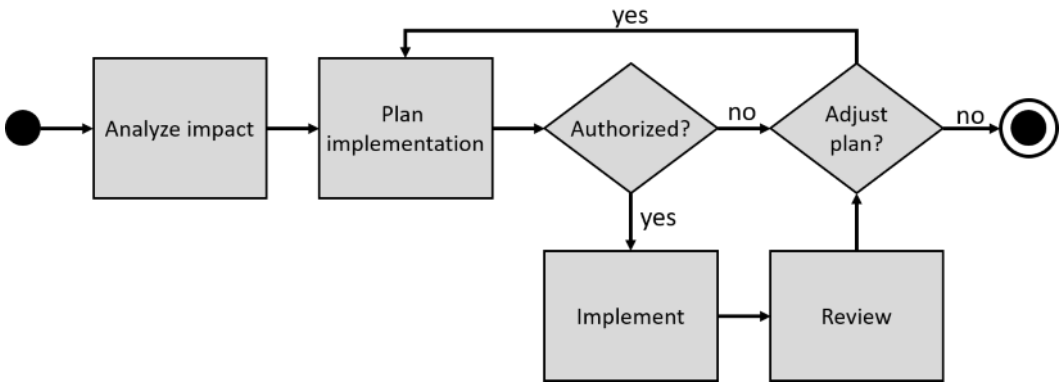


Figure 6.5 Change enablement process

As long as a requirement is in a draft state, the author has the authority to change it and no strict process is followed.

As soon as a requirement is released for further use in the project, the author is no longer free to decide, as every change will have an impact on other work products based on this requirement. Before deciding whether a change should be implemented, an impact analysis should be performed to clarify the efforts and risks of the change. This is where traceability is indispensable. In a *linear* development approach, the change authority will often be assigned to project management, a steering committee, or a *Change Control Board*, and a process is followed, with a formal decision on the change and the planning of its implementation. In an *iterative* development approach, the change authority usually lies with the product owner, who decides on the change and adds an accepted change to the product backlog as a new item (work product). The further implementation is then handled just like any other product backlog item.

Once a requirement is implemented in an operational system, an even stricter process should be followed, as every change will now influence users and business processes.

Here, a distinction is often made between *standard* (low-risk, well understood, and pre-authorized, e.g., a change to the VAT percentage), *normal* (based on a formal Request for Change, scheduled, assessed, and authorized, e.g., a change to a price calculation algorithm), and *emergency* changes (to be implemented as soon as possible, e.g., to resolve an incident—but that seldomly involves a change of requirements). Usually, the change authority lies with a *Change Advisory Board* [Math2019]; in an iterative approach like DevOps, a change may be authorized by a release manager.

6.8 Prioritization

Requirements themselves are just concepts in the minds of people. They bring value only when they are implemented in an operational system. This implementation takes effort, time, money, and attention. In most cases, these resources are limited, which means that not all requirements can be implemented, at least not at the same time. This in turn means that the stakeholders have to decide which requirements should come first and which could be implemented later (or not at all). In other words: prioritization [Wieg1999].

The priority of a requirement is defined as the level of importance assigned to it according to certain criteria [Glin2020]. Consequently, you first have to determine what criteria should be used to assess the requirements before you can prioritize them. However, before you can determine the assessment criteria, you must know what the goal of the prioritization is. That goal is usually not your goal as a Requirements Engineer but the goal of certain stakeholders, so you must decide who the stakeholders are for this prioritization. And when you know their goal, it will usually be clear that not all requirements will have to be prioritized but rather only a defined subset.

Summarizing the above, we can outline a sequence of steps to be followed if we want to prioritize requirements:

- Define major goals and constraints for the prioritization

  Project and system context largely determine the reasons for prioritization. If, for instance, you prioritize to decide which features will be implemented in the next release, you might focus on business value; if the goal is to select user stories for the next iteration, story points and technical dependencies would be more prominent. Technical or legal constraints might limit the choices to be made.

- Define desired assessment criteria

  In principle, the goals and constraints dictate the criteria to be used. Commonly used criteria are business value for stakeholders, urgency perceived by users, effort to implement, risks for usage, logical and technical dependencies, the legally binding nature of a requirement, or just the (inter–) subjective preference of relevant stakeholders. Sometimes only a single criterion is used but a balanced selection of several relevant criteria may yield a better outcome.

- Define the *stakeholders* that have to be involved

  Goals and constraints influence which stakeholders you should involve in the prioritization but on the other hand, certain stakeholders themselves set these goals, so you must be aware of the interdependency. As an example, when prioritizing for the launch of a new system, you would probably invite business representatives and a panel of future customers. When prioritizing the product backlog to decide on the next iteration, the scrum team would be involved.

- Define the *requirements* that have to be prioritized

  It is unlikely that the whole set of requirements has to be prioritized. Once again, this depends primarily on the goals and constraints for prioritizing. For instance, constraints may dictate certain requirements to be must-haves. In fact, it is only useful to prioritize requirements for which you have a choice whether or not to include them in a next step of the development process. This means that the project phase is also an important factor. In an early phase, you might include draft versions in the prioritization; in a late phase, you will often restrict prioritization to requirements that are in a stable version. Be aware that requirements to be prioritized should be at a comparable level of abstraction depending on the prioritization goals. In an early project phase, for instance, you might prioritize themes or features while prioritizing user stories at iteration planning.

- Select the prioritization *technique*

  A prioritization technique is the way in which your stakeholders prioritize the requirements. As described below, there are several techniques, which differ in effort, thoroughness, and level of detail. Here too, goals and constraints set the stage, but the most important factor is that the stakeholders involved agree on the technique that they intend to use. If not, they will not accept the outcome and your prioritization effort is in vain.

- Perform prioritization

  When all preparation has been done, the actual prioritization can be performed. First, all defined assessment criteria shall be applied to all selected requirements. Together with the stakeholders involved, you then apply the selected prioritization technique to the requirements assessed. As a result, you get a prioritized list of requirements. However, there might be a problem. Different stakeholders might have different priorities, even if they agree on the criteria assessed. In that case, you typically have a requirements conflict that should be resolved just like any other conflict as described in Section 4.3 on conflict resolution.

Taking a closer look at prioritization techniques, we distinguish between two categories:

- Ad hoc techniques

  With ad hoc techniques, experts assign priorities to the selected requirements based on their own experience. In principle, this prioritization is based on a single criterion, being the subjective perception of the expert. If this expertise is at a high level and acceptable to the stakeholders, such a technique can be a quick, cheap, and easy way to achieve prioritization. A variant would be to invite several experts and calculate some kind of average priorities. Common ad hoc techniques include Top-10 ranking and MoSCoW (Must have, Should have, Could have, Won't have this time) prioritization. Kano analysis (Section 4.2.1) is also useful: the dissatisfiers are must-

haves, the satisfiers should-haves, and the delighters can be could- or won't-haves. For more background, see, for example, [McIn2016].

- *Analytical* techniques

  Analytical techniques employ a systematic process for assigning priorities. In such techniques, experts assign weights to multiple assessment criteria (such as benefit, cost, risk, time to implement, etc.) and subsequently, requirements priorities are calculated as weighted outcomes based on these criteria. Such techniques take more effort and time but have the advantage of giving a clear insight into the factors that determine the priorities and into the process by which the priorities are established. This can stimulate the acceptance of the outcome among the stakeholders. However, two aspects must be kept in mind. First, the outcome is heavily influenced by the weight factors that are used in the calculation of the result. Therefore, an agreement among the stakeholders about these weight factors must be established before the actual prioritization. Otherwise, some might try to change the weight factors in order to manipulate the priorities. The second aspect to consider is that the criteria assessed are mostly estimates, not measured facts. And the estimates are often on a simple ordinal scale such as low, medium, high. Thus, the quality of the estimates is decisive for the quality of the resulting prioritization. Nevertheless, analytical techniques are useful for providing a clearly underpinned prioritization that is understood and thus accepted by the stakeholders involved. For a detailed explanation of analytical techniques, see [Olso2014].

It may be tempting to apply detailed, thorough techniques and spend a lot of time producing perfectly accurate estimates in terms of money, hours, expected sales numbers, etc. This could result in requirement A having a calculated priority of 22.76, requirement B of 23.12, and requirement C of 20.29. You would then conclude that evidently, C must be done first and A prior to B. However, you have probably just introduced a pseudo-accuracy with this calculation, and it would be better to conclude that those three requirements are equally important, which might have been your gut feeling right from the start.
Always make sure that the effort you spend in prioritizing is justified by the value of a correct prioritization itself. So once again, keep the goals in mind and remember Principle 1: value orientation.

6.9 Further Reading

The textbooks by Pohl [Pohl2010], Davis [Davi2005], Hull, Jackson and Dick [HuJD2011], van Lamsweerde [vLam2009] and Wiegers and Beatty [WiBe2013] provide a comprehensive overview of requirements management. Additional insights to the topic of requirements management is consolidated in the CPRE Advanced Level handbook for Requirements Management by Bühne and Herrmann [BuHe2019].

Cleland-Huang, Gotel and Zisman [ClGZ2012] provide an in-depth treatment of traceability.

Olson [Olso2014] and Wiegers [Wieg1999] deal with prioritization techniques.

7 Tool Support

A Requirements Engineer needs tools to practice his craftsmanship properly—just as a carpenter needs his tools, pencil, a hammer, saw, and drill to design and realize a piece of furniture. Without tools, it is difficult or impossible to record the requirements, work together on the requirements, and be in control of the requirements.

This chapter examines the different types of Requirements Engineering (RE) tools available and the aspects that need to be taken into account to introduce Requirements Engineering tools into an organization.

7.1 Tools in Requirements Engineering

Requirements Engineering is a difficult task without the support of tools. Tools are needed to support Requirements Engineering tasks and activities. Existing tools focus on supporting specific tasks, such as documenting requirements or supporting the RE process, and rarely on all tasks and activities in the Requirements Engineering process. It is therefore not surprising that the Requirements Engineer must have a set of tools at his disposal to support the various components in the Requirements Engineering process—just as the carpenter needs several tools (e.g., computer-aided design (CAD)) to design a piece of furniture and needs tools like a saw, scraper, and sandpaper to realize it.

Tools are just an aid to the Requirements Engineering process and the Requirements Engineer, and such tools are called CASE (computer-aided software engineering) tools. CASE tools support a specific task in the software production process [Fugg1993].

We differentiate between different types of tools that support the following aspects of Requirements Engineering:

- Management of requirements

  Tools in this category have the properties needed to support the activities and topics described in Chapter 6. With these kinds of tools, more control can be established over the Requirements Engineering process. Requirements are subject to change and in an environment where this happens frequently, a tool with the relevant properties is indispensable. Tools in this category support:

  - Definition and storage of requirements attributes to identify and collect data about work products and requirements as described in Section 6.5
  - Facilitation and documentation of the prioritization of requirements (Section 6.8)
  - Life cycle management, version control, configurations and baselines as described in Sections 6.2, 6.3, and 0
  - Tracking and tracing of requirements, as well as defects in the requirements and work products (Section 6.6)
  - Change management for requirements; as we learned in Section 6.7, changes are inevitable and have to be carefully managed

- Requirements Engineering process

  To support the Requirements Engineering process, information is needed to allow the process to be adjusted or improved. This kind of tool can:

  - Measure and report on the Requirements Engineering process and workflow
  - This information helps to improve the Requirements Engineering process and reduces waste.
  - Measure and report on the product quality
  - This information helps to find defects and flaws, which in turn can be used to improve product quality.
- Documentation of knowledge about the requirements

  The amount of knowledge (and requirements) built up in a project can be enormous. In addition, a large amount of knowledge is built up about a product during its life cycle. All the relevant information must be carefully documented to enable the following:

  - Sharing and creation of a common understanding of the requirements
  - Securing the requirements as a legal obligation
  - An overview of and insight into the requirements
- Modeling of requirements

  As we learned in Section 3.4.1.6, expressing requirements in both diagrams and natural language uses the strengths of both forms of notation. A tool that can model requirements allows you to:

  - Structure your own thoughts; it can be used as an aid to thinking
  - Specify the requirements in a more formal language than textual requirements, with all benefits that brings
- Collaboration in Requirements Engineering

  When several people and disciplines work on the same project, a tool can support and enable this collaboration, especially in the world in which we now live, where more and more activities are performed locally (at home). This kind of tool supports the elicitation, documentation, and management of requirements.

- Testing and/or simulation of the requirements

  Tools are becoming more and more sophisticated. More and more options are being developed for testing and/or simulating requirements in advance. This allows a better prediction of whether the proposed requirements will have the intended effect.

The tools available are often a mix of the above. As mentioned before, different tools may need to be combined to adequately support Requirements Engineering. If different tools are used, it is important to pay attention to the integration between them and the interaction with other applications and systems in order to ensure smooth operation.

Sometimes, other kinds of tools (for example, office or issue-tracking tools) are used, or rather, misused, to document or manage requirements. However, these tools have their

limitations and should be used only when the Requirements Engineers and stakeholders are in control of the RE process and requirements are aligned. Otherwise, this is a major risk in the RE process, as such tools do not support any requirements management activities.

7.2 Introducing Tools

Selecting an RE tool is no different to selecting a tool for any other purpose. You should describe the objectives, context, and requirements before selecting and implementing the appropriate tool(s).

Tools are just an aid to the Requirements Engineering process and the Requirements Engineer. They do not solve organizational or human issues. Imagine that, together with your colleagues, you want to document the requirements in a uniform manner. Tools can support this—for instance, with a template in a word processing tool or wiki page. This does not ensure that all your colleagues adopt this working method, neither does it ensure that your colleagues have the discipline to record and manage their requirements in this way. What can help is to make agreements with each other, to check whether the agreements are being fulfilled, and to be able to communicate with each other if agreements are not adhered to. A tool is not going to help you with this. Introducing a Requirements Engineering tool requires clear Requirements Engineering responsibilities and procedures.

A tool can help you to configure your Requirements Engineering process effectively and efficiently. Tools often provide a framework based on best practices experience. These frameworks can then be tailor-made to suit the situation.

As we have learned in the previous chapters, core Requirements Engineering activities are not stand-alone processes.

Selecting the appropriate RE tools starts with the definition of the objectives and/or problems you want to solve in the RE process. The next step is to determine the context of the system (in this case, the tool set). Consider the aspects of the context—i.e., stakeholders, processes, events, etc., and apply your Requirements Engineering skills to specify the requirements for the RE tools. Practice what you preach.

The next sections describe some of the aspects that have to be taken into account when introducing a (new) Requirements Engineering tool into your organization.

7.2.1 Consider All Life Cycle Costs beyond License Costs

The most obvious costs, such as purchase costs or licensing costs, are usually factored in. In addition, less visible costs must also be taken into account, such as the use of resources in the implementation, operation, and maintenance of the tool.

7.2.2 Consider Necessary Resources

Specifying the requirements and supervising the selection process requires the necessary resources, in addition to the costs mentioned in the previous section. People necessary to

guide the selection process, Requirements Engineers, hardware resources, and other resources should not be overlooked. After the tool has been put into use, resources may also be required for maintenance and user support.

7.2.3 Avoid Risks by Running Pilot Projects

The introduction of a new tool can threaten the control over the current requirements base. A requirements chaos can arise because there is a transition from the old working method and/or tools to the new working method and tools. Introduction of a new tool during an existing project will irrevocably lead to a delay in the delivery of the requirements and even the project.

The introduction of a new tool, possibly with a different working method, should be tested on a small scale, where the risks and impact remain manageable. There are several ways to do this:

- Apply the tool to a non-critical project/system
- Use the tool redundantly alongside an existing project
- Apply the tool to a fictional situation/project
- Import/copy the requirements of a project that has already been completed

When you have reached the point where the tool meets the set goals and requirements, it can be rolled out more widely within the organization or other projects.

7.2.4 Evaluate the Tool according to Defined Criteria

Selecting the appropriate tool can be a difficult task. Extensive verification of whether the objectives and requirements are met is a standard approach in Requirements Engineering. A systematic approach that assesses the tool from different perspectives also contributes to making the right choice. The following perspectives can be considered:

- Project perspective

  This point of view highlights the project management aspects. Does the tool support the project and the information required in the project?

- Process perspective

  This perspective verifies the support of the Requirements Engineering process. Does the tool sufficiently support the RE process? Can it be sufficiently adapted to the existing RE process and working method?

- User perspective

  This perspective verifies the degree of application by the users of the tool. This is an important view because if users are not satisfied with the tool, the risk of the tool not being accepted increases. Does the tool sufficiently support the authorization of users and groups? Is it sufficiently user-friendly and intuitive?

- Product perspective

  The functionalities offered by the tool are verified from this angle. Are the requirements sufficiently covered by the tool? Where is the data stored? Is there a roadmap with the functional extensions for the tool? Is the tool still supported by the supplier for the time being?

- Supplier perspective

  With this perspective the focus lies on the service and reliability of the supplier. Where is the supplier located? How is the support for this tool arranged?

- Economic perspective

  This perspective looks at the business case: does the tool deliver sufficient benefits in relation to the costs? What are the (management) costs for the purchase and maintenance? What does the tool provide for the RE process? Is a (separate) maintenance contract required?

- Architecture perspective

  This perspective assesses how the tool fits into the (IT) organization. Does the technology applied suit the organization? Can the tool be sufficiently linked with other systems? Does the tool fit into the IT landscape and does it comply with the architectural constraints?

7.2.5 Instruct Employees on the Use of the Tool

Once a tool has been selected, the users should become familiar with the opportunities the tool can add to the Requirements Engineering process. The users—i.e., the Requirements Engineers—should be trained in how to use the tool in the existing Requirements Engineering process. If the users are not sufficiently trained, this may mean that not all the benefits of the tool are used. In fact, it is possible that the tool will be used incorrectly, with all the associated consequences.

The Requirements Engineering process can also be changed due to the tool selected. Aspects in the Requirements Engineering process that were not possible before can be made possible with a new tool: for example, adequate version management, modeling of requirements, etc. This can mean that new procedures are agreed, templates are adapted or applied, changes are made to the working method, and so on. The involvement of the Requirements Engineer in this change contributes to the success of the tool's acceptance.

7.3 Further Reading

The following literature can be consulted for an overview of available tools and tool evaluations. Juan M. Carrillo de Gea et. al. provide a comprehensive overview of the role of Requirements Engineering tools [dGeA2011].

The article by Barbara Kitchenham, Stephen Linkman, David Law [KiLL1997] describes and validates a method for systematic tool evaluation. If you are searching for an RE tool, a comprehensive list of tools for Requirements Engineering is provided on the Volere website [Vole2020] or at [BiHe2020].

8 References

[Alex2005] Ian F. Alexander: A Taxonomy of Stakeholders – Human Roles in System Development. International Journal of Technology and Human Interaction 2005, 1(1), 23–59.

[AnPC1994] Annie I. Antón, W. Michael McCracken, Colin Potts: Goal Decomposition and Scenario Analysis in Business Process Reengineering. CAiSE (Conference on Advanced Information Systems Engineering), 1994, 94–104.

[Armo2004] Philip G. Armour. The Laws of Software Process: A New Model for the Production and Management of Software. Boca Raton, Fl.: CRC Press, 2004.

[Axelos2019] Axelos: ITIL Foundation: ITIL 4 Edition. Axelos Ltd., 2019.

[BaBo2014] Stéphane Badreau, Jean-Louis Boulanger: Ingénierie des Exigences. Paris: Dunod, 2014 (in French).

[BeeA2001] Kent Beck et al.: Principles behind the Agile Manifesto. http://agilemanifesto.org/principles.html, 2001. Last visited August 2022.

[BiHe2020] Andreas Birk, Gerald Heller: List of Requirements Management Tools. https://makingofsoftware.com/resources/list-of-rm-tools/, 2020, Last visited August 2022.

[Boeh1981] Barry W. Boehm: Software Engineering Economics, Englewood Cliffs, New Jersey: Prentice Hall, 1981.

[BoRJ2005] Grady Booch, James Rumbaugh, Ivar Jacobson: The Unified Modeling Language User Guide, 2nd edition. Reading, MA: Addison-Wesley, 2005.

[Bour2009] Lynda Bourne: Stakeholder Relationship Management – A Maturity Model for Organisational Implementation. Farnham: Gower, 2009.

[BuHe2019] Stan Bühne, Andrea Herrmann: Handbook Requirements Management according to the IREB Standard – Education and Training for the IREB Certified Professional for Requirements Engineering Qualification Advanced Level Requirements Management. Karlsruhe: IREB. https://www.ireb.org/downloads/#cpre-advanced-level-requirements-management-handbook, 2019. Last visited August 2022.

[CaDJ2014] Dante Carrizo, Oscar Dieste, Natalia Juristo: Systematizing Requirements Elicitation Technique Selection. Information and Software Technology 2014, 56(6), 644–669.

[Chen1976] Peter P.-S. Chen: The Entity-Relationship Model: Toward a Unified View of Data, ACM Transactions on Database Systems 1976, 1(1), 9–36.

[ClGZ2012] Jane Cleland-Huang, Olly Gotel, Andrea Zisman (eds.): Software and Systems Traceability. London: Springer, 2012.

[Cock2001] Alistair Cockburn: Writing Effective Use Cases. Boston: Addison-Wesley, 2001.

[Cohn2004] Mike Cohn: User Stories Applied: For Agile Software Development. Boston: Addison-Wesley, 2004.

[Cohn2010] Mike Cohn: Succeeding with Agile: Software Development Using Scrum. Upper Saddle River, NJ: Addison-Wesley, 2010.

[CoWe1998] Reidar Conradi, Bernhard Westfechtel: Version Models for Software Configuration Management. ACM Computing Surveys 1998, 30(2), 232–282.

[DaTW2012] Marian Daun, Bastian Tenbergen, Thorsten Weyer: Requirements Viewpoint. In: K. Pohl, H. Hönninger, R. Achatz, M. Broy: Model-Based Engineering of Embedded Systems, Heidelberg: Springer, 2012.

[Davi1993] Alan M. Davis: Software Requirements – Objects, Functions, and States. 2nd Edition, Englewood Cliffs, New Jersey: Prentice Hall, 1993.

[Davi1995] Alan M. Davis: 201 Principles of Software Development. New York: McGraw-Hill, 1995.

[Davi2005] Alan M. Davis: Just Enough Requirements Management – Where Software Development Meets Marketing. New York: Dorset House, 2005.

[DeBo2005] Edward De Bono: De Bono's Thinking Course (Revised Edition), Barnes & Noble Books, 2005.

[DeCo2007] Design Council: 11 Lessons: A Study of the Design Process. https://www.designcouncil.org.uk/resources/report/11-lessons-managing-design-global-brands, 2007. Last visited August 2022.

[dGeA2011] Juan M. Carrillo de Gea, Joaquín Nicolás, José L. Fernandez-Alemán, Ambrosio Toval, Christof Ebert, Aurora Vizcaíno: Requirements Engineering Tools. IEEE Software 2011, 28(4), 86–91.

[DeMa1978] Tom DeMarco: Structured Analysis and System Specification. New York: Yourdon Press, 1978.

[DIN66001] DIN 66001:1983-12: Information processing; graphical symbols and their application. Deutsches Institut für Normung e.V., Berlin, 1983 (in German).

[Eber2014] Christof Ebert: Systematisches Requirements Engineering, 5. Auflage. Heidelberg: dpunkt 2014 (in German).

[Fowl1996] Martin Fowler: Analysis Patterns: Reusable Object Models. Reading, MA: Addison-Wesley, 1996.

[FLCC2016] Xavier Franch, Lidia Lopez, Carlos Cares, Daniel Colomer. (2016). The i* Framework for Goal-Oriented Modeling. In Domain Specific Conceptual Modeling, Springer, 485-506.

[Fugg1993] Alfonso Fuggetta: A Classification of CASE Technology. IEEE Computer 1993, 26(12), 25–38.

[GaWe1989] Donald C. Gause and Gerald M. Weinberg: Exploring Requirements: Quality before Design. New York: Dorset House, 1989.

[GFPK2010] Tony Gorschek, Samuel Fricker, Kenneth Palm, and Steven A. Kunsman: A Lightweight Innovation Process for Software-Intensive Product Development. IEEE Software 2010, 27(1), 37–45.

[GGJZ2000] Carl A. Gunter, Elsa L. Gunter, Michael Jackson, Pamela Zave: A Reference Model for Requirements and Specifications. IEEE Software 2000, 17(3), 37–43.

[GHJV1994] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Pattern – Elements of Reusable Object-Oriented Software. Reading, Mass.: Addison-Wesley, 1994.

[Gilb1988] Tom Gilb: Principles of Software Engineering Management. Reading, Mass.: Addison Wesley, 1988.

[Glas1999] Friedrich Glasl: Confronting Conflict – A First-Aid Kit for Handling Conflict. Stroud, Gloucestershire: Hawthorn Press, 1999.

[GlFr2015] Martin Glinz and Samuel A. Fricker: On Shared Understanding in Software Engineering: An Essay. Computer Science – Research and Development 2015, 30(3–4), 363–376.

[Glin2007] Martin Glinz: On Non-Functional Requirements. 15th IEEE International Requirements Engineering Conference, Delhi, India, 2007, 21–26.

[Glin2008] Martin Glinz: A Risk-Based, Value-Oriented Approach to Quality Requirements. IEEE Software 2008, 25(2), 34–41.

[Glin2016] Martin Glinz: How Much Requirements Engineering Do We Need? Softwaretechnik-Trends 2016, 36(3), 19–21.

[Glin2019] Martin Glinz: Requirements Engineering I. Course Notes, University of Zurich, 2019. https://www.ifi.uzh.ch/en/rerg/courses/archives/hs19/re-i.html#resources. Last visited August 2022.

[Glin2020] Martin Glinz: A Glossary of Requirements Engineering Terminology. Version 2.0. https://www.ireb.org/downloads/#cpre-glossary, 2020. Last visited August 2022.

[GlWi2007] Martin Glinz and Roel Wieringa: Stakeholders in Requirements Engineering (Guest Editors' Introduction). IEEE Software 2007, 24(2), 18–20.

[GoFi1994] Orlena Gotel, Anthony Finkelstein: An Analysis of the Requirements Traceability Problem. 1st International Conference on Requirements Engineering, Colorado Springs, 1994, 94–101.

[GoRu2003] Rolf Goetz, Chris Rupp: Psychotherapy for System Requirements. 2nd IEEE International Conference on Cognitive Informatics (ICCI'03), London, 2003, 75–80.

[Gott2002] Ellen Gottesdiener: Requirements by Collaboration: Workshops for Defining Needs, Boston: Addison-Wesley Professional, 2002.

[GreA2017] Eduard C. Groen, Norbert Seyff, Raian Ali, Fabiano Dalpiaz, Joerg Doerr, Emitzá Guzmán, Mahmood Hosseini, Jordi Marco, Marc Oriol, Anna Perini, Melanie Stade: The Crowd in Requirements Engineering – The Landscape and Challenges. IEEE Software 2017, 34(2), 44–52.

[Greg2016] Sarah Gregory: "It Depends": Heuristics for "Common Enough" Requirements. Keynote speech at REFSQ 2016, Essen, Germany, 2016.

[GRL2020] Goal oriented Requirement Language. University of Toronto, Canada https://www.cs.toronto.edu/km/GRL. Last visited August 2022.

[GrSe2005] Paul Grünbacher, Norbert Seyff: Requirements Negotiation. In A. Aurum, C. Wohlin (eds.): Engineering and Managing Software Requirements. Berlin: Springer, 2005, 143–162.

[Hare1988] David Harel. On Visual Formalisms. Communications of the ACM 1988, 31(5), 514–530.

[HoSch2020] Stefan Hofer, Henning Schwentner: Domain Storytelling — A Collaborative Modeling Method. Available from Leanpub, http://leanpub.com/domainstorytelling. Last visited August 2022.

[HuJD2011] Elizabeth Hull, Ken Jackson, Jeremy Dick: Requirements Engineering. 3rd ed., Berlin: Springer: 2011.

[Hump2017] Aaron Humphrey: User Personas and Social Media Profiles. Persona Studies 2017, 3(2), 13–20.

[IEEE830] IEEE Recommended Practice for Software Requirements Specifications. IEEE Std 830–1998, 1998.

[ISO19650] ISO 19650. Organization and Digitization of Information about Buildings and Civil Engineering Works, including Building Information Modelling (BIM)– Information Management Using Building Information Modelling – Part 1 and 2, 2018.

[ISO5807] ISO/IEC/IEEE 1985–02: Information processing; Documentation symbols and conventions for data, program and system flowcharts, program network charts and system resources charts. International Organization for Standardization, Geneva, 1985.

[ISO25010] ISO/IEC/IEEE 25010:2011: Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. International Organization for Standardization, Geneva, 2011.

[ISO29148] ISO/IEC/IEEE 29148: Systems and Software Engineering – Life Cycle Processes – Requirements Engineering. International Organization for Standardization, Geneva, 2018.

[Jack1995] Michael Jackson: Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices. New York: ACM Press, 1995.

[Jack1995b] Michael Jackson: The World and the Machine. 17th International Conference on Software Engineering 1995 (ICSE 1995), 287–292.

[Jaco1992] Ivar Jacobson: Object-oriented software engineering: a use case driven approach. New York: ACM Press, 1992.

[JaSB2011] Ivar Jacobson, Ian Spence, Kurt Bittner: Use Case 2.0: The Guide to Succeeding with Use Cases. Ivar Jacobson International SA, 2011.

[KiLL1997] Barbara Kitchenham, Stephen Linkman, David Law: DESMET: A Methodology for Evaluating Software Engineering Methods and Tools. Computing & Control Engineering Journal 1997, 8(3), 120–126.

[KSTT1984] Noriaki Kano, Nobuhiku Seraku, Fumio Takahashi, Shinichi Tsuji: Attractive Quality and Must-Be Quality. Hinshitsu (Quality – Journal of the Japanese Society for Quality Control) 1984, 14(2), 39–48 (in Japanese).

[Laue2002] Søren Lauesen: Software Requirements: Styles and Techniques. London: Addison-Wesley, 2002.

[LaWE2001] Brian Lawrence, Karl Wiegers, and Christof Ebert: The Top Risks of Requirements Engineering. IEEE Software 2001, 18(6), 62–63.

[LiOg2011] Jeanne Liedtka, Tim Ogilvie: Designing for Growth: A Design Thinking Tool Kit for Managers. New York: Columbia University Press, 2011.

[LiSS1994] Odd I. Lindland, Guttorm Sindre, Arne Sølverg: Understanding Quality in Conceptual Modeling. IEEE Software 1994, 11(2), 42–49.

[LiSZ1994] Horst Lichter, Matthias Schneider-Hufschmidt, Heinz Züllighoven: Prototyping in Industrial Software Projects – Bridging the Gap Between Theory and Practice. IEEE Transactions on Software Engineering 1994, 20(11), 825–832.

[LiQF2010] Soo Ling Lim, Daniele Quercia, Anthony Finkelstein: StakeNet: Using Social Networks to Analyse the Stakeholders of Large-Scale Software Projects. 32nd International Conference on Software Engineering (ICSE 2010), 2010, 295–304.

[MaGR2004] Neil Maiden, Alexis Gizikis, Suzanne Robertson: Provoking Creativity: Imagine What Your Requirements Could Be Like. IEEE Software 2004, 21(5), 68–75.

[Math2019] Joseph Mathenge: Change Control Board vs Change Advisory Board: What's the Difference? https://www.bmc.com/blogs/change-control-board-vs-change-advisory-board, Nov. 22, 2019. Last visited August 2022.

[McIn2016] John McIntyre: MoSCoW or Kano Models – How Do You Prioritize? https://www.hotpmo.com/management-models/moscow-kano-prioritize, Oct. 20, 2016. Last visited August 2022.

[MNJR2016] Walid Maalej, Maleknaz Nayebi, Timo Johann, and Guenther Ruhe: Toward Data-Driven Requirements Engineering. IEEE Software 2016, 33(1), 48–54.

[Moor2014] Christopher W. Moore: The Mediation Process – Practical Strategies for Resolving Conflicts, 4th edition. Hoboken, NJ: John Wiley & Sons, 2014.

[MWHN2009] Alistair Mavin, Philip Wilkinson, Adrian Harwood, and Mark Novak: Easy Approach to Requirements Syntax (EARS). 17th IEEE International Requirements Engineering Conference (RE'09), Atlanta, Georgia, 2009, 317–322.

[NuKF2003] Bashar Nuseibeh, Jeff Kramer, Anthony Finkelstein: ViewPoints: Meaningful Relationships are Difficult! 25th International Conference on Software Engineering (ICSE'03), Portland, Oregon, 2003, 676–681.

[OleA2018] K. Olsen et al.: Certified Tester, Foundation Level Syllabus – Version 2018. International Software Testing Qualifications Board, 2018.

[Olso2014] David Olson: Matrix Prioritization. http://www.bawiki.com/wiki/Matrix-Prioritization.html, 2014. Last visited August 2022.

[OMG2013] Object Management Group: Business Process Model and Notation (BPMN), version 2.0.2. OMG document, formal/2013-12-09. https://www.omg.org/spec/BPMN/. Last visited August 2022.

[OMG2017] Object Management Group: OMG Unified Modeling Language (OMG UML), version 2.5.1. OMG document, formal/2017-12-05. https://www.omg.org/spec/UML/About-UML/. Last visited August 2022.

[OMG2018] Object Management Group: OMG Systems Modeling Language (OMG SysML™), version 1.6. OMG document, ptc/2018-12-08. https://www.omg.org/spec/SysML/About-SysML/. Last visited August 2022.

[Osbo1948] Alex F. Osborn: Your Creative Power: How to Use Imagination. C. Scribner's Sons, 1948. (Accessed as digital reprint: Read Books Ltd. (epub eBook), April 2013).

[Pich2010] Roman Pichler: Agile Product Management with Scrum – Creating Products that Customers Love, Boston: Addison-Wesley, 2010.

[Pohl2010] Klaus Pohl: Requirements Engineering: Fundamentals, Principles, and Techniques. Berlin-Heidelberg: Springer, 2010.

[PoRu2015] Klaus Pohl, Chris Rupp: Requirements Engineering Fundamentals: A Study Guide for the Certified Professional for Requirements Engineering Exam, (2nd ed). Rocky Nook, Santa Barbara, 2015.

[Rein1997] Donald G. Reinertsen: Managing the Design Factory – A Product Developer's Toolkit. The Free Press, 1997.

[Rein2009] Donald G. Reinertsen: The Principles of Product Development Flow: Second Generation Lean Product Development. Redondo Beach, Ca.: Celeritas Publishing, 2009.

[Ries2011] Eric Ries: The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses. New York: Crown Business, 2011.

[Robe2001] S. Ian Robertson: Problem Solving. Hove, East Sussex: Psychology Press, 2001.

[RoRo2012] Suzanne Robertson and James Robertson: Mastering the Requirements Process: Getting Requirements Right. 3rd edition. Boston: Addison-Wesley, 2012.

[RuJB2004] James Rumbaugh, Ivar Jacobson, Grady Booch: The Unified Modeling Language Reference Manual, 2nd edition. Reading, MA: Addison Wesley, 2004.

[Rupp2014] Chris Rupp: Requirements-Engineering und Management, 6. Auflage. München: Hanser, 2014 (in German).

[SoSa1998] Ian Sommerville and Pete Sawyer: Requirements Engineering: A Good Practice Guide. Chichester: John Wiley & Sons, 1997.

[SwBa1982] William Swartout and Robert Balzer: On the Inevitable Intertwining of Specification and Implementation. Communications of the ACM 1982, 25(7), 438-440.

[Verd2014] Dave Verduyn: Discovering the Kano Model, in: Kano model, https://www.kanomodel.com/discovering-the-kano-model, 2014. Last visited August 2022.

[vLam2009] Axel van Lamsweerde: Requirements Engineering: From System Goals to UML Models to Software Specifications. Chichester: John Wiley & Sons, 2009.

[Vole2020] Volere Requirements Resources: https://www.volere.org. Last visited August 2022.

[WiBe2013] Karl Wiegers and Joy Beatty: Software Requirements. 3rd edition. Redmond, Wa.: Microsoft Press, 2013.

[Wieg1999] Karl E. Wiegers: First Things First: Prioritizing Requirements. https://www.processimpact.com/articles/prioritizing.pdf, 1999. Last visited August 2022.

[ZoCo2005] Didar Zowghi, Chad Coulin: Requirements Elicitation: A Survey of Techniques, Approaches, and Tools. In A. Aurum, C. Wohlin (eds.) Engineering and Managing Software Requirements. Berlin: Springer, 2005, 19-46.