

Peter Hruschka
Kim Lauenroth
Markus Meuten
Gareth Rogers
Stefan Gärtner
Hans-Jörg Steffe

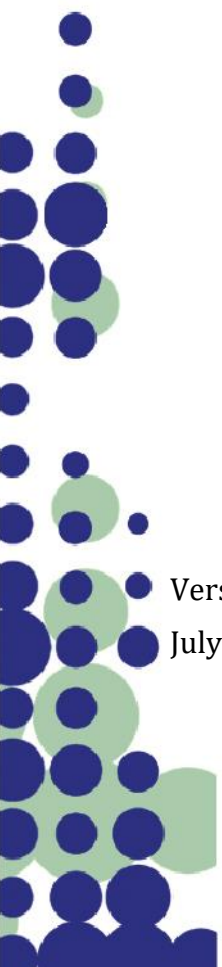
Handbook RE@Agile

Education and Training
for

IREB Certified Professional for Requirements Engineering

Advanced Level RE@Agile

Practitioner | Specialist



Version 2.0.0

July 2022

Terms of Use

This Handbook, including all of its parts, is protected by copyright law. With the consent of the copyright owners and following copyright law, the use of the Handbook is permitted—unless explicitly mentioned it is not permitted. This applies in particular to reproductions, adaptations, translations, microfilming, storage and processing in electronic systems, and public disclosure.

Training providers may use this Handbook as a basis for seminars and training provided that the copyright holder is acknowledged and the source and owner of the copyright is mentioned. In addition, with the prior consent of IREB, this Handbook may be used for advertising purposes.

Any individual or group of individuals may use this Handbook as a basis for study, articles, books or other derived publications provided that the copyright holder is acknowledged and the source and owner of the copyright is mentioned.

Acknowledgements

This handbook was initially created in 2018 by Bernd Aschauer, Peter Hruschka, Kim Lauenroth, Markus Meuten, Gareth Rogers.

Our thanks to Rainer Grau for his intensive reviews of the RE@Agile syllabus, to all reviewers of this document as well as Stefan Sturm, Sibylle Becker and Ruth Rossi for their encouragement and support. Review comments were provided by Sacha Reis and Sven van der Zee. Proofreading by Michiel van der Voort.

Approved for release on February 18, 2022, by the IREB Council upon recommendation of Xavier Franch.

We thank everybody for their involvement.

Copyright © 2017-2022 for this Handbook is with the authors listed above. The rights have been transferred to the IREB International Requirements Engineering Board e.V.

Table of Contents

Terms of Use.....	2
Acknowledgements	2
Table of Contents.....	3
Foreword	6
Version History	7
1. What is RE@Agile.....	8
1.1 History of Requirements Engineering and Agility.....	8
1.2 Learning from each other.....	11
1.3 RE@Agile – A Definition	13
2. A Clean Project Start.....	15
2.1 Visions and Goals.....	15
2.1.1 Fundamentals.....	15
2.1.2 Techniques for Vision/Goal Specification	17
2.1.3 Changing Vision and/or Goals.....	21
2.1.4 Specifying the System Boundary Fundamentals	21
2.1.5 Documentation of the System Boundary	24
2.1.6 The Inevitability of a Changing Scope.....	28
2.2 Stakeholder Identification and Management.....	28
2.2.1 Fundamentals.....	28
2.2.2 Identification of Stakeholders	29
2.2.3 Management of Stakeholders	31
2.2.4 Sources for Requirements beyond Stakeholder	32
2.3 Summary	32
2.4 Case Study and Exercises	33
3. Handling Functional Requirements	35
3.1 Different Levels of Requirements Granularity	35
3.2 Communicating and Documenting on Different Levels.....	37
3.3 Working with User Stories.....	41
3.3.1 The 3 C model.....	41
3.3.2 A template for user stories:.....	42

3.3.3	INVEST: Criteria for “good” stories	43
3.3.4	Supplementing stories with other requirements artifacts.....	43
3.4	Splitting and Grouping Techniques.....	44
3.5	Knowing When to Stop.....	46
3.6	Project and Product Documentation of Requirements.....	47
3.7	Summary	49
4.	Handling Quality Requirements and Constraints.....	50
4.1	Understanding the Importance of Quality Requirements and Constraints.....	51
4.2	Adding Precision to Quality Requirements.....	53
4.3	Quality Requirements and Backlog.....	57
4.4	Making Constraints Explicit	57
4.5	Summary	59
5.	Prioritizing and Estimating Requirements	61
5.1	Determination of Business Value.....	61
5.2	Business Value, Risk.....	63
5.3	Expressing Priorities and Ordering the Backlog	64
5.4	Estimating User Stories and other Backlog Items.....	68
5.5	Choosing a Development Strategy.....	72
5.6	Summary	76
6.	Scaling RE@Agile.....	77
6.1	Scaling Requirements and Teams.....	77
6.1.1	Organizing large scale requirements.....	79
6.1.2	Organizing Teams.....	80
6.1.3	Organizing Lifecycles/Iterations.....	82
6.2	Criteria for structuring Requirements and Teams in the Large.....	83
6.2.1	Product-focused backlog	83
6.2.2	Self-organizing teams and collaborative decision-making	84
6.2.3	Understanding feature-based requirements splitting	85
6.2.4	Considerations when feature-based requirements splitting is not possible	86
6.2.5	Telecoms company example	87
6.3	Roadmaps and Large Scale Planning.....	90
6.3.1	Representing roadmaps.....	91
6.3.2	Synchronizing teams with roadmaps.....	94

6.3.3	Developing roadmaps	95
6.3.4	Validating roadmaps	96
6.4	Product Validation	96
	List of Abbreviations	98
	References	99

Foreword

This *Handbook* complements the syllabus of the CPRE Advanced Level RE@Agile.

This Handbook is intended for training providers who want to offer seminars or training on RE@Agile Practitioner and/or Specialist according to the IREB standard. It is also aimed at training participants and interested parties who want to get a detailed insight into the content of this advanced level module. It can also be used when applying Requirements Engineering methods in an agile environment according to the IREB standard.

This Handbook is not a substitute for training on the topic. The Handbook represents a link between the Syllabus (which lists and explains the learning objectives of the module) and the broad range of literature that has been published on the topic.

The contents of this Handbook, together with references to more detailed literature, support training providers in preparing training participants for the certification exam. This Handbook provides training participants and interested parties an opportunity to deepen their knowledge of Requirements Engineering in an agile environment and to supplement the detailed content based on the literature recommendations. In addition, this Handbook can be used to refresh existing knowledge about the various topics of RE@Agile, for instance after having received the RE@Agile Practitioner or the RE@Agile Specialist certificate.

Suggestions for improvements and corrections are always welcome!

E-mail contact: info@ireb.org

We hope that you enjoy studying this Handbook and that you will successfully pass the certification exam for the IREB Certified Professional for Requirements Engineering Advanced Level RE@Agile - Practitioner - or the IREB Certified Professional for Requirements Engineering Advanced Level RE@Agile - Specialist.

More information on the IREB Certified Professional for Requirements Engineering Advanced Level module RE@Agile can be found at: <http://www.ireb.org>.

Stefan Gärtner

Peter Hruschka

Kim Lauenroth

Markus Meuten

Gareth Rogers

Hans-Jörg Steffe

Version History

Version	Date	Comment	Author
1.0.0	October 10, 2018	Initial Version	Bernd Aschauer, Peter Hruschka, Kim Lauenroth, Markus Meuten and Gareth Rogers
1.0.1	September 11, 2019	Minor improvements (typos, formatting, a few inconsistencies removed) in the context of the translation to German.	Markus Meuten Hans-Jörg Steffe Ruth Rossi
1.0.2	December 17, 2019	Consistent usage of the term refinement meeting and product backlog refinement.	Hans-Jörg Steffe
2.0.0	July 1, 2022	Complete reorganization of the chapter 6; Consistent design of the Figures; Bug fixing in chapter 1-5 (e.g. replaced “minimal” with “minimum” in minimum viable product and minimum marketable product, replaced “development team” with “developers”); inclusion of the Advanced Level split in Practitioner and Specialist	Peter Hruschka, Kim Lauenroth, Markus Meuten, Gareth Rogers, Stefan Gärtner, Hans-Jörg Steffe

1. What is RE@Agile

Good Requirements Engineering is a recognized success factor for product or system development, regardless of the development methodology applied.

In this chapter you will get an understanding of the background and history of Requirements Engineering and of the background and history of agile approaches (chapter 1.1). You will learn why sometimes these two disciplines are considered to be incompatible – which is a popular misconception. You will learn that – despite their history – techniques and methods from the Requirements Engineering discipline are being used (without a clear reference to its origin) in specific development approaches (like Waterfall and Scrum). You will also learn that agile approaches (like Scrum, Lean Development and Kanban) need good requirements practices to deliver successful products and systems.

In chapter 1.2 we will discuss the strengths and weaknesses of Requirements Engineering methods and of agile approaches. While Requirements Engineering emphasizes the importance of eliciting, understanding and documenting key stakeholders' requirements in order not to build the wrong product or system, most agile approaches emphasize the importance of trustful cooperation among the stakeholders. In agile, frequent feedback loops based on visible results are used to avoid wrong assumptions being made or periods of misunderstanding lasting too long.

IREB developed the advanced module RE@Agile to combine the strengths of both disciplines. As you can guess: the goals of Requirements Engineering AND agile approaches are not in conflict, but they rather complement each other – if applied correctly!

The final chapter 1.3 introduces IREB's definition of RE@Agile. In a nutshell you will learn how your development projects can benefit from this integrated approach.

1.1 History of Requirements Engineering and Agility

Based on their respective histories, Requirements Engineering and agile approaches are often considered separately rather than together. Let us consider some key milestones in these histories to better understand how this situation arose. These milestones are captured in overview in Figure 1. (note that they were chosen by the authors of this Handbook to emphasize important sources for the advanced level module. We do not claim to have captured the complete history of development methods).

In the late 1970s the term “software crisis” resounded throughout the IT-community. The most important complaint: Product development is a complex process and often the products do not satisfy the users. The answer of scientists and methodologists was the waterfall model (originally suggested by Winston Royce, but made popular by Barry Boehm). One of its remedies for the software crisis was to introduce a “Requirements Phase” before designing, building and testing systems. Its goal was to reach agreement among important stakeholders on what the product was intended to do before building it.

Requirements specifications at that time were mostly documents with natural language. Around the same time (mid to end 1970s) many suggestions were made to use graphical models in addition to text with the aim to improve the precision of requirements and to avoid inconsistencies. In 1975 Peter Chen suggested Entity-Relationship models to capture business relevant data. In 1978/1979 Douglas Ross and Tom DeMarco introduced Structured Analysis and Design Technique (SADT) to capture business functionality.

In the mid 1980s Barry Boehm formulated the “Spiral Model”. Requirements Engineering became an iterative technique, while introducing risk management and more frequent feedback cycles.

From a method and notation point of view, 1992 was an important milestone for Requirements Engineering: Ivar Jacobson proposed a “Use Case Driven Approach”. The ideas of emphasizing “actors” (or users) in the context of the system, and of thinking end-to-end across the whole product, were not new.

McMenamin/Palmer (in 1984) and Hammer/Champy, in their “Business Process Reengineering” Methodology, also emphasized this sort of process thinking. But the notation of Ivar Jacobson – simple stick figures and ellipses, supported by natural language descriptions of these use cases – became very popular.

Another important milestone for Requirements Engineering was the Unified Process of Ivar Jacobson – made popular as the “Rational Unified Process” (RUP). RUP recognizes Requirements Engineering as a “discipline” instead of a “phase”. This discipline spans all of the phases (with varying degrees of emphasis).

All modern process models have adopted this distinction between disciplines (like business analysis, requirements, design, implementation, testing) and phases (like inception, elaboration, construction, transition – in the RUP-terminology). The latter allow for manageable milestones, while the former ensure that appropriate techniques and practices are established for ongoing work.

The international standardization of UML (Unified Modeling Language) in 1997 by the OMG (Object Management Group) helped to make requirements specifications using use case models, activity diagrams, state charts and so on more popular, especially since many tools supported these notations.

Thinking in terms of end-to-end business processes was further enhanced by the standardization of the BPMN (Business Process Model and Notation). While Ivar Jacobson’s use cases were often misinterpreted to be “just the IT part of the business processes”, BPMN models are closer to the “business”. This addressed one important RE-issue: the alignment of business and IT.

Another important aspect of Requirements Engineering has been discussed as early as 1986 with HP’s introduction of FURPS: the importance of quality requirements. FURPS (Functionality, Usability, Reliability, Performance and Supportability) was one of the first approaches to emphasize quality aspects in addition to functionality.

This was refined by the ISO/IEC standard 9126, which established many additional categories of qualities to be achieved by systems. The latest revision of this standard is the ISO/IEC standard 25010 (also known as SQuaRE – Systems and Software Quality Requirements and Evaluation), in which the importance of security in modern systems is emphasized.

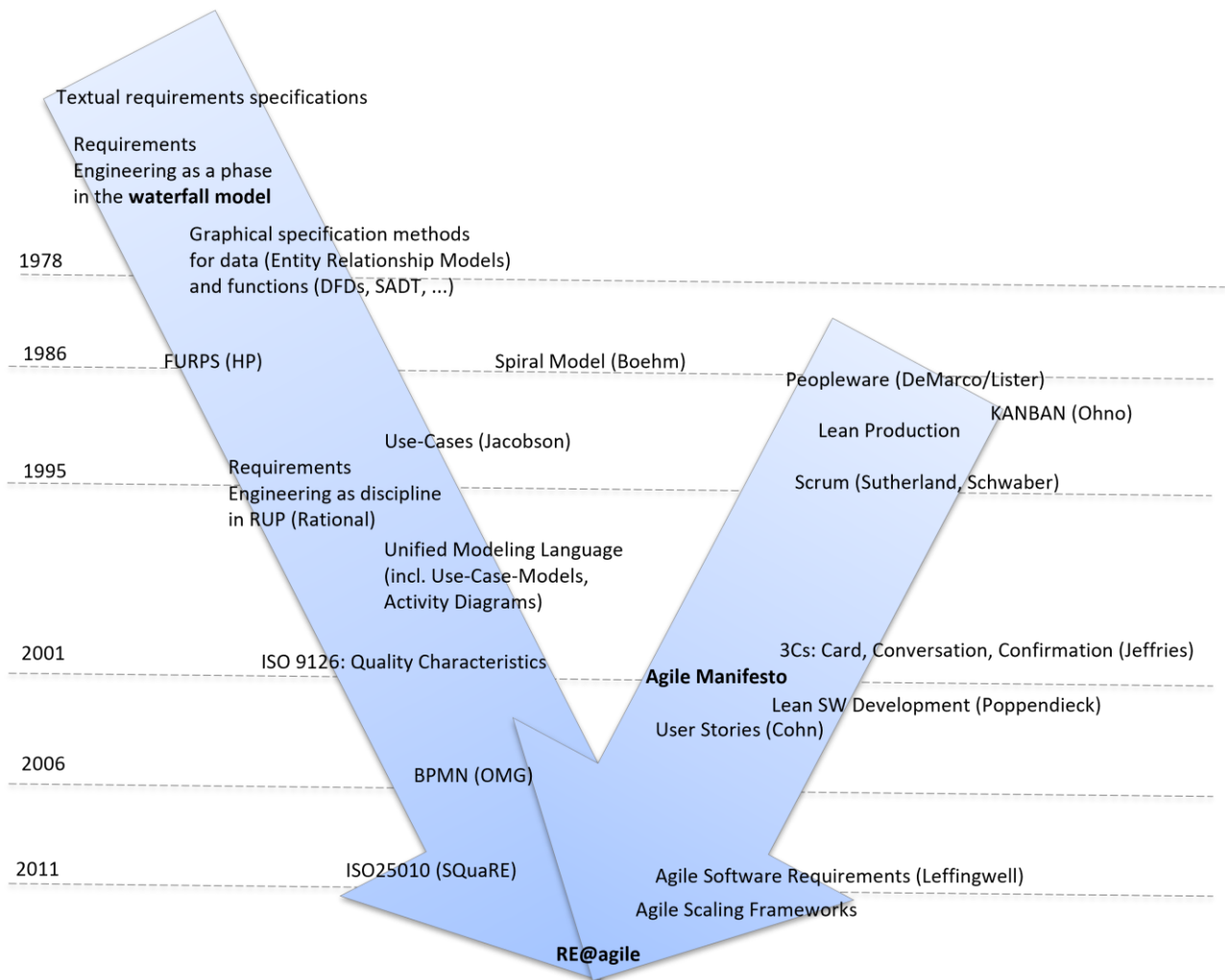


Figure 1: Selected milestones in RE and Agile

Some key ideas of agile approaches were published long before the Agile Manifesto appeared in 2001.

Tom DeMarco and Tim Lister coined the term “Peopleware” in 1987 to emphasize the importance of human cooperation and teams.

Toyota published success stories involving Kanban and Lean Manufacturing (or Lean Production) in the late 1980s. Both concepts (Kanban and Lean) are core ideas in today’s agile methods.

Scrum, a “framework for developing and sustaining complex products”, was first published by Mike Beedle and Ken Schwaber in 1995. It introduces the role of a “Product Owner”, responsible for the product’s success within an organization. The Product Owner¹ sets the priorities of requirements (often called epics or stories). Partly due to its simplicity (3 roles, 4 artifacts, 5 meetings), Scrum became very popular around the world.

In 2001 a group of 17 individuals representing popular approaches like Extreme Programming, Scrum, DSDM, Adaptive Software Development, Crystal, Feature-Driven Development and Pragmatic Programming met in Utah and agreed on a “manifesto”. The Agile Manifesto, as it became known, shifted the emphasis of system development from contracts, documents and long-term planning and processes to cooperation, openness to change and feedback based on frequent releases.

¹ For the sake of brevity, we will use the role name “product owner” in this Handbook, whenever we refer to a person responsible for managing the requirements. For a definition see the glossary.

In the same year Ron Jeffries – one of the signees of the Agile Manifesto– published the 3C model (Card, Conversation, Confirmation), to distinguish “social” user stories from “documentary” requirements practices such as use cases.

A few years later Mike Cohn suggested a format for these cards: user stories. User stories emphasize three important issues: Who wants what why. (As a <type of user> I want <some goal> so that <some reason>).

Since Scrum was mainly developed for smaller teams (up to ten people), more and more scaling frameworks (for example SAFe, LeSS, DAD...) were published from 2010 onwards, suggesting ways of cooperating in larger or distributed teams.

Dean Leffingwell [Leffingwell2010] coined the term “Agile Software Requirements” by publishing a book in 2011 with this title, which led to the term “Agile Requirements Engineering”. Although this term became popular for performing Requirements Engineering tasks according to the principles of the Agile Manifesto, there is a danger that it may lead to the misunderstanding that there are two ways of Requirements Engineering: classical Requirements Engineering and Agile Requirements Engineering. IREB’s view is that there is only good or bad Requirements Engineering - in a non-agile or agile world. Therefore, we call the IREB approach RE@Agile.

1.2 Learning from each other

Agile and Requirements Engineering are two disciplines with different origins and distinct goals that can nevertheless learn a lot from each another.

Let us start with some key Requirements Engineering ideas and see how they can benefit from the agile mindset. After that we will look at some basic agile principles and discuss how Requirements Engineering techniques can further improve them.

IREB defines Requirements Engineering as a systematic and disciplined approach to the system specification with the following goals:

1. Knowing the relevant requirements, achieving consensus among the stakeholders about these requirements, documenting them according to given standards, and managing them systematically;
2. Understanding and documenting the stakeholder’s desires and needs;
3. Specifying and managing requirements, to minimize the risk of delivering a system that does not meet the stakeholders’ desires and needs.

The first point – knowing relevant requirements before plunging into a solution – is undisputed. But agility is very explicit about how “relevant” should be interpreted: just in time! Not all requirements are relevant at the beginning of an endeavor. A vision statement or a set of goals are sufficient to get started. Before parts of the solution are developed a thorough understanding of this subset of requirements is necessary. Others – that are not so urgent for the business – can be left more vague and can be refined later.

A goal of Requirements Engineering is to achieve consensus among stakeholders. Who could challenge this goal? Agility suggests achieving the consensus by intensive, trustful collaboration: intensive discussions about requirements should take place until all stakeholders share the feeling that they are well understood. Another agile mechanism to achieve consensus among the stakeholders, is quick feedback through demonstrable product increments. In many environments seeing a (partial) product increment and being able to use it, is more successful in finding open issues than creating large volumes of precise documents that are often not read.

Requirements Engineering strives to document stakeholder's desires and needs. Agility warns us that we should not produce documents for the sake of producing documents. Documenting requirements (in an adequate form for the stakeholders) should either (1) support the process of achieving consensus or (2) satisfy externally imposed constraints (for instance legal constraints or traceability requirements) or (3) make life easier for defining requirements for the next release without being forced to start from scratch.

The last goal in the definition of Requirements Engineering is managing requirements, to minimize the risk of delivering a system that does not meet the stakeholders' desires and needs. To achieve this, agility suggests constantly checking the priority and estimates of backlog items (the requirements).

The principles of agility help to refocus Requirements Engineering in terms of its efficiency, flexibility and collaborative nature. Conversely, there are many insights of Requirements Engineering from which agile approaches can also benefit.

Agility strongly encourages trustful collaboration and communication among all relevant stakeholders. In many agile methods this usually means frequent and open verbal communication between clients and users on one side (those that have needs or requirements) and developers on the other (those that can provide solutions to the needs and requirements).

While trustful communication is an excellent way to achieve a joint understanding of requirements, this is by far not the only way to elicit requirements. Requirements Engineering has developed an extensive body of knowledge on elicitation techniques (for example [RoRo2013]) suitable for use in different environments and under particular constraints. For example: creativity techniques, such as brainstorming, help to create product backlog items quickly in innovative projects; product archeology can create quicker results when working on new versions of existing products; questionnaires may help to get feedback quickly from a large number of widely dispersed stakeholders that you would never get into one meeting room.

Agile Product Owners can benefit greatly from having a range of such elicitation techniques at their disposal and picking a suitable subset that helps to fill the product backlog more quickly than "just talking".

By focusing on trustful communication agile approaches often downplay the importance of precise documentation. User story approaches, in particular, emphasize that the cards to denote stories are principally a reminder of the discussion, and not a replacement for exact requirements (also see chapter 3.3). We agree that plain natural language (in contrast to more formal requirements notations) is often an adequate way to understand each other. However natural language is sometimes not precise enough to avoid misinterpretation. Many other requirements notations have been developed over the last decades – including many graphical notations – that allow stakeholders to overcome the lack of precision of natural language. Some business processes might be more easily discussed using activity diagrams, data-flow diagrams or Business Process Model and Notation (BPMN) than by writing cards for the steps of the process. Some objects to be dealt with are sometimes more easily sketched using information models, and some state-driven systems could benefit from state models to clarify which activities should be performed in which state. Once again, Product Owners and developers should know such notations – not for the sake of applying a formalism, but rather for shortening discussions.

Another agile credo is delivering working software frequently, that is working iteratively and creating a series of product increments. It does not, however, make sense to start with iterative development if the team is not aligned on a vision or a set of goals. For a single Scrum Product Owner in full command of a product it may be easy to have a vision or set of goals. If, however, the Product Owner has to coordinate with a number of "important" stakeholders, then stakeholder analysis, goal alignment and scope definition should precede any detailed requirements work. These activities are included within the idea of a "clean project start" introduced in chapter 2.

Summarizing the thoughts of this chapter: agility helps us to create a culture for successful product development and helps Requirements Engineering become more flexible, efficient and collaborative. From a requirements point of view the cornerstones of agility are trustful cooperation of all stakeholders and striving for short term incremental results. While techniques such as user stories sketched on story cards work well, there are many other elicitation and validation techniques, developed over decades of Requirements Engineering research, that can make Product Owners and their development teams even more productive – if, of course, applied correctly and without formalistic exaggeration.

In the RE@Agile Primer [Primer2017] we concluded: “The most important value is shared by Requirements Engineering and agile, and that is to make the end user of the product happy because the solution fits their needs or cures their greatest pains.”

In this Advanced Level Module we will go into detail to show how ideas from both worlds can be used together to achieve this goal. In the following definition of RE@Agile we first find it helpful to set out our own guiding principles for the rest of this Handbook.

1.3 RE@Agile – A Definition

RE@Agile is a cooperative, iterative and incremental approach with four goals:

1. Knowing the relevant requirements at an appropriate level of detail (at any time during system development);
2. Achieving sufficient agreement about the requirements among the relevant stakeholders;
3. Capturing (and documenting) the requirements according to the constraints of the organization;
4. Performing all requirements-related activities according to the principles of the Agile Manifesto.

As mentioned above we will use the Scrum terminology of a Product Owner as the role that is responsible for the cooperative approach and therefore as the role responsible for good Requirements Engineering.

Let us explore the key ideas of this definition in detail:

1. RE@Agile is a cooperative approach:
“Cooperative” emphasizes the agile idea of intensive stakeholder interaction by frequently inspecting the product, providing feedback on it and adapting and clarifying the requirements as by working together everyone can learn and achieve more.
2. RE@Agile is an iterative process:
This suggests the idea of “just in time”-requirements. Requirements do not have to be complete before starting technical design and implementation. Stakeholders can iteratively define (and refine) those requirements that should be implemented soon at the appropriate level of detail.
3. RE@Agile is an incremental process:
Implementation of those requirements that are considered to deliver highest business value or reduce the highest risks form the first increment. Early increments strive to create a minimum viable product (MVP) or a minimum marketable product (MMP). From then on, the next increments can be added to that product, constantly picking the ones that promise the highest business value, thus constantly increasing the business value of the resulting product.

The first goal (“relevant requirements known at the appropriate level of detail”) again refers to the iterative approach: “relevant” are those requirements that should be implemented soon. And those have to be understood very precisely (including their acceptance criteria) – especially by the developers.

They have to conform to the “definition of ready” (DoR). Other requirements – that are not highest priority yet – can be kept at a higher abstraction level – only to be detailed further as soon as they become more important.

The prerequisite for the second goal (“sufficient agreement among relevant stakeholders”) is to know all stakeholders and their relevance for the system being developed. The person responsible for requirements (usually the Product Owner) has to negotiate the requirements with those relevant stakeholders to determine the order of their implementation.

Agile approaches value intensive and ongoing communication about requirements over communication about documentation. Nevertheless, the third goal emphasizes the importance of documentation at an appropriate level of detail (which differs from situation to situation). Organizations may have to keep documentation about requirements (for instance for legal purposes, traceability or maintenance). In these cases, agile approaches have to ensure that the appropriate documentation was produced. However, it does not have to be created upfront. It might save time and effort to create the documentation in parallel to the implementation, or even after the implementation. It might also be useful to create some artifacts like data models, activity models or state models as temporary documentation to support the discussion about requirements.

Requirements management summarizes all activities to be done once you have existing requirements and requirements related artifacts. In agile most requirements management activities are included in the constant refinement process of the backlog items. But classical requirement management also includes attribution of requirements, version management, configuration management as well as traceability among requirements and traceability to other development artifacts. RE@Agile suggests to minimize these efforts or to balance efforts and benefits:

- ▶ Extensive version management can be replaced by quick iterations leading to product increments (for instance the change-history of requirements since they were first encountered is less interesting than their current valid state);
- ▶ Configuration management (base lining) is included in the iterative determination of sprint backlogs, i.e., grouping those requirements that currently promises the highest business value.

Therefore, some of the time (and paper-) consuming requirements management activities of non-agile approaches are substituted by activities based on agile principles. And some others are well supported by tools that help to automatically keep track of relationships between requirements and about history without additional human effort.

The next chapters of this Handbook will discuss various aspects of RE@Agile in more detail.

Chapter 2 will discuss the prerequisites for successful system development: balancing vision and/or goals, stakeholders and scope of the system.

Chapter 3 and 4 will discuss handling of functional requirements, quality requirements and constraints on different levels of granularity.

Chapter 5 will discuss the process of estimating, ordering and prioritizing requirements to determine the sequence of increments.

The chapters 2 through 5 mainly emphasize handling requirements by a group of developers (of 3 – 9 persons).

Chapter 6 discusses scaling Requirements Engineering for larger, potentially distributed teams, including overall product planning and road mapping.

2. A Clean Project Start

Preparing the workshop before starting a major project is an established tradition in many crafts. It includes preparing and gathering the necessary material, cleaning and sorting the tools, removing waste from the previous project, and agreeing on the cornerstones of the upcoming project. Because of the immaterial nature of software, such behavior may appear inadequate or old-fashioned. The opposite is in fact true.

Most of the work in software development is mental work or plain thinking. This means that most of the work is not visible from the outside compared to traditional crafts. In a workshop or a construction site, mistakes are often visible to others and can therefore be corrected immediately. A mistake in thinking can only be noticed if the output of the thinking is visible in some form. The output may then be recognized by a person or system as wrong, leading to the understanding or identification of the mistake in the thinking.

Agile approaches are often not aware of this problem. People think that direct communication and fast feedback cycles are sufficient. Although they are really helpful and valuable, they are not sufficient. For example: if the shared big picture and other visible artifacts are missing when the development starts, then direct communication and fast feedback cycles cannot prevent multiple reworks.

The idea of a clean project start presented in this chapter describes important prerequisites that enable successful iterative, incremental system development.

You will learn that a Clean Project Start should consist of three activities producing three tangible results that can be used to steer iterative work:

- ▶ Definition of the vision and/or goals of the system
- ▶ Identification of the currently known scope of the system and the system boundary
- ▶ Identification of relevant stakeholders and other important requirements sources

You will learn details for each activity and their corresponding techniques in the next chapters. At the end of this chapter, we will present the case study iLearnRE including exercises to practice the clean project start. The case study will be used as an ongoing example for additional exercises in the following chapters.

2.1 Visions and Goals

2.1.1 Fundamentals

The product vision and/or the goals of the product are of the utmost importance of every development activity. They set the overall direction for development and guide all other activities. Vision and/or goals are either triggered by problems or unsatisfactory circumstances encountered in the current environment, or by changes in the environment that force us to react (for instance the introduction of new legislation), or by innovative ideas that promise more or better business.

We use both terms –vision and goals– interchangeably. Agile methods often prefer to talk about vision while Requirements Engineering approaches usually use the word “goals”. Both can be considered as the most abstract formulation of what should be achieved by the system. All team members and all relevant stakeholders should be aware of the defined vision and goals to understand what the team is striving for.

The Product Owner is responsible for the formulation of the vision and/or goals. The Product Owner is also responsible for explaining the details to team members. Being responsible does not mean that the Product Owner must define the vision or goals alone. Typically, the Product Owner discusses the vision and/or goals with relevant stakeholders and collects their input and feedback.

A common pitfall when defining a vision and/or goals is choosing the wrong perspective, meaning formulating a vision and/or goals that say something about the system under development or part of it. An example is the following statement:

“Create a website for buying and reading electronic books and audio books.”

This vision describes a system (a website) for buying and reading electronic books. Depending on the circumstances, developing such a system may be a good idea or not. However, this statement is far too restrictive to become a good vision statement because it characterizes the system rather than stating what should be achieved by developing the system. The following statement chooses a different perspective:

“Sell electronic/audio books to people in every place in the world (with an Internet connection) and allow them to read/listen to the acquired book instantaneously.”

This statement is better compared to the previous one for several reasons:

1. The statement defines what the system shall achieve instead of defining the function of the system.
2. The statement focuses on the benefits of the system for people (and the users).
3. Buying electronic/audio books wherever they are and reading/listening to the book immediately.
4. The statement does not predefine the type of system.
5. A website may not be the proper solution for reading electronic books/listening to audio books.

The major drawback of formulating visions and goals that focus on the system itself rather than on the what the impact is of the system, is that such formulations restrict the solution space for the team right from the very beginning of the project. As a rule of thumb, avoid any reference to the system under development (and the word “system” itself) in a vision or goal statement.

Visions and goals are normally associated with a time horizon. This time horizon defines the period in which a vision or goal should be achieved. We therefore recommend that the definition of visions and goals should always have a period (or even a specific date) attached to it. It is not necessary to include the period in the formulation of the vision or goal itself, but the period should be clear to all team members and stakeholders.

Agile recommends the definition of visions and/or goals for each iteration. Therefore, there can be different statements for different time periods. A system or product development could have long term goals (or strategic visions), for instance for the next 3 years, which in turn can be broken down into goals to be achieved in specific years. And of course, in iterative development one should also have goals to be achieved in the next iteration.

The benefit of defining visions or goals with a long-term perspective is, that the team members and all stakeholders have a clear understanding of the big picture, and of the timeframe in which the big picture will be achieved. This benefit can be illustrated with the bookshop example presented earlier. The stated vision could be sub-divided as follows:

Overall vision “Sell electronic and audio books to people in every place in the world (with an Internet connection) and allow them to read/listen to the acquired book instantaneously.”

- ▶ End of month 6: Sell electronic books to people in every place in the world and allow them to read the electronic book immediately.
- ▶ End of year 1: Sell audio books to people in every place on the world and allow them to listen to the electronic book immediately.
- ▶ End of year 2: Sell combined electronic and audio books to people in every place in the world and allow them to read and listen to the electronic book at the same time immediately.

The sub-divided vision presents a clear timeframe for the project and includes the important information that there will be a bundle of electronic and audio books where the reader can both listen to and read the text at the same time. This information is very important for the team since they should design the system for reading electronic books in such a way that it is possible to include the audio book later in the process. Furthermore, the team is now able to give feedback to answer the question: Is it realistic to realize the three goals within the defined timeframe?

2.1.2 Techniques for Vision/Goal Specification

In the previous chapter, you have seen fundamentals related to the definition of vision and/or goals. In this chapter, you will learn specific techniques that can support you in the development and definition of vision and/or goals. Whatever form is chosen: every stakeholder has the right to know what the team is striving for. Therefore, the definition of the vision and the initial goals must take place at the beginning of a development effort.

2.1.2.1 SMART

SMART is an acronym and refers to a simplified style of writing goals and objectives, proposed in 1981 by George T. Doran [Doran1981].

According to Doran, the acronym stands for:

- ▶ Specific – target a specific area for improvement;
- ▶ Measureable – quantify or at least suggest an indicator of progress;
- ▶ Assignable – specify who will do it;
- ▶ Realistic – state what results can realistically be achieved, given the available resources;
- ▶ Time-related – specify when the result(s) can be achieved.

This original definition has been adapted by the Agile community in various ways. From a Requirements Engineering perspective, the following definition is appropriate:

- ▶ Specific – target a specific area for improvement;
- ▶ Measureable – quantify or at least suggest an indicator of progress;
- ▶ Achievable (instead of assignable) – state a goal that is achievable for the team;
- ▶ Relevant (instead of realistic) – state a goal that is relevant for the stakeholders;
- ▶ Time-bound – specify when the result(s) can be achieved.

This modification takes into account two ideas behind agile development:

1. Goals should focus on achievability by the team without focusing on resources. Resources are not planned; they are assigned by prioritization.
2. Relevance of the goal, meaning the value that is attached to the goal, is more important than the question of achievability with respect to available resources.

To illustrate the application of SMART, we use the example given above:

“Sell electronic and audio books to people in every place in the world (with an Internet connection) and allow them to read/listen to the acquired book instantaneously”.

The SMART criteria are satisfied by this statement as follows:

Criterion	Example
Specific	The experience of buying and consuming electronic and audio books will be improved
Measureable	The measurable outcome is buying electronic and audio books at every place in the world (with an Internet connection) and reading/listening to them instantaneously
Achievable	Internet and mobile technology can provide the desired result
Relevant	Electronic and audio books are a popular medium for many people
Time-bound	The timeframe is detailed (see above for details)

The SMART criteria can be applied either as a template or as a checklist for a goal formulation. In the template approach, you explicitly describe each element of the SMART criteria. The table above is an example of this approach. The disadvantage of the template approach is that it typically creates redundancy in the formulation.

In the checklist approach, you use the SMART criteria to verify if your goal statement covers all aspects.

A good combination of both approaches is the following: Make up your mind by using the SMART template and then use the outcome to define a precise goal that can be communicated easily.

2.1.2.2 PAM

PAM is an alternative set of criteria for goal formulation proposed by [Robertson2003]. The criteria are defined as follows:

- ▶ What is the purpose (P)?
- ▶ What is the business advantage (A)?
- ▶ How would we measure (M) that advantage?

The PAM criteria focus on the business value behind a goal and exclude the time-perspective of the SMART criteria. A benefit of using this approach at an early stage is that it focuses on the different values instead of forcing a time-perspective into the goal definition.

Referring again to our example above, the PAM criteria are not completely satisfied, as shown by the following table:

Criterion	Example
Purpose	The experience of buying and consuming electronic and audio books will be improved
Business Advantage	Not stated explicitly
Measure	The measurable outcome is buying electronic and audio books at every place in the world (with an Internet connection) and reading/listening to them instantaneously

The business advantage is not clear in our example. A business advantage could be:

- ▶ People buy more electronic or audio books when they are available instantaneously;
- ▶ People buy more electronic or audio books when they are traveling since the books are available all over the world.

Like the SMART criteria, the PAM criteria can be used as a template or as a checklist for goal formulations.

2.1.2.3 Product Vision Box

The idea behind SMART and PAM is the definition of explicit criteria that support the wording of goals. These criteria are useful when you have gathered much information and want to structure this information into proper goals and/or a proper vision.

Another way of approaching the definition of visions and goals is the Product Vision Box introduced by [Highsmith2001]. The idea behind the Product Vision Box is that you create a *physical package* for your product that shows the key benefits and ideas of a product to potential customers in a store.

A common format of the Product Vision Box is a half-day workshop. Invite key stakeholders, if possible from the whole spectrum of those involved with your product (for example end users, marketing, technical staff).

You provide cardboard boxes, various types of material (for example paper, pencil, crayons, board markers, aluminum foil, wires) and media material (for example newspapers, magazines, photos) to the participants of the workshop.

The agenda of the workshop should consist of alternating building and presentation phases. During the building phase, a team of workshop participants (3-4 people) creates one or more boxes (packages).

During the presentation phase, the boxes are presented without any explanation to the participants. Every participant can make up his or her mind about each box. Afterwards, the creators present the ideas behind the box (es) and a discussion takes place.

As an option people that were not part of the workshop can be invited to join the last presentation phase. This way external feedback is provided to the group, reducing the effect of group thinking.

The main advantage of the Product Vision Box is that people think about the product idea from the final outcome backwards. A product package typically provides information about the most important key features or benefits of a product. Such an approach implicitly supports a focus on goals and the overall vision. It is great fun for the participants, since it creates a tangible outcome that can be used later on as a kind of reference point for further discussion.

A common objection against the Product Vision Box is that there are types of products that cannot be sold in simple packages but can be developed agile. An example from the field of non-IT projects: Organizational Change projects have to deal with various problem domains and needs and therefore multi-dimensional solutions have to be created that will not fit into one box.

2.1.2.4 News from the future

Another technique to approach the formulation of vision and goals is to write a newspaper article about your product that comes from the future (see [HeHe2010]). This technique is derived from techniques for personality development that motivate people to think about their life from the end, for instance by writing their own obituary.

The news from the future can cover various topics and headlines. Good starting points can be:

- ▶ Successful product presentation – write an article from the perspective of a journalist who participated at your successful product presentation. Mention features, impressions or ideas that this journalist found great about your product;
- ▶ Happy 10th anniversary – imagine that your product celebrates its 10th anniversary and that a journalist writes about this event in a newspaper article. Mention ups and downs in the story of your product and how it has had an impact on peoples’ lives or on the business you are in;
- ▶ Product crash report – imagine your product fails and that a journalist reports on its failure. Mention the reasons that led to the failure, and think about gaps in your knowledge of the customer, missing features, or other quality problems.

The resulting article can be analyzed to identify potential vision and goals. It is also a good starting point for further activities. For example, the SMART or the PAM criteria might subsequently be used to create precise vision and/or goals statements.

The news-from-the-future technique can be performed by individuals or can be done as a group exercise during a workshop. In the group exercise, the participants should write rather short articles that can be read and discussed during the workshop.

2.1.2.5 Vision Boards

The term “Vision Board” refers to a class of techniques that provide structured graphical representation of the vision and/or the goals on a physical board. The general idea is that:

- a) The board provides a content- or time-oriented structure to visualize the whole set of vision and/or goals to the stakeholders;
- b) The vision board is considered to be a living entity that is modified constantly to represent the current understanding of all stakeholders;
- c) The vision board is the single point of truth for all stakeholders when it comes to the vision and/or goals.

A very simple example of a vision board consists of three columns:

- ▶ Short-term vision and related short-term goals: what do we want to achieve in the near future (for instance 4 weeks)?
- ▶ Mid-term vision and related mid-term goals: what do we want to achieve in the mid-term (for instance 6 months)?
- ▶ Long-term vision and related long-term goals: what do we want to achieve in the long-term (for instance 3 years)?

A second, structure-oriented example of a vision board is the “Product Vision Board”, defined by [Pichler2016]. It consists of the following elements:

- ▶ Vision: What is your motivation for creating the product? Which positive change should it bring about?
- ▶ Target group: Which market or market segment does the product address? Who are the target customers and users?
- ▶ Needs: What problem does the product solve? Which benefit does it provide?
- ▶ Product: What product is it? What makes it stand out? Is it feasible to develop the product?

- ▶ Business goals: How will the company benefit from this product? What are the business goals?

2.1.2.6 Canvas Techniques

The term “Canvas Technique” refers to a set of techniques that aim at providing a structured overview of several aspects of a product. Canvas Techniques are close to Vision Board techniques, but typically have a broader scope and do not solely focus on the vision and/or goals of the product. Nevertheless, the vision and/or goals are always part of canvases and are developed in conjunction with the other aspects covered by the canvas.

Because of this broader scope, there are more slots when using Canvas Techniques than when using a vision board. Therefore, Canvas Techniques require more space to document all aspects. Hence, the term canvas is used, because a canvas can be much larger than a board. Nevertheless, the general idea behind Canvas Techniques is similar to vision boards.

A popular example of Canvas Technique is the “Business Model Canvas” from [OsPi2010]. The idea behind it is to describe a company’s or product’s value proposition, infrastructure, customers and finances.

Another example is the Opportunity Canvas from [Patton2014]. This canvas assumes an already existing product that has to be improved.

2.1.3 Changing Vision and/or Goals

Goals may change during a development effort because of new stakeholders or because of a changed understanding of the system or the context. Therefore, the documentation of the vision and/or goals should be updated on a regular basis. Techniques such as the Vision Board provide a physical representation of the vision and/or goals and allow for easy communication of changed goals.

Changes to the vision or the goals should be documented explicitly including the rationale for changing them. Formal documentation of these changes is not necessary. Lightweight ways of documenting changes are:

- ▶ A diary or journal (analogue or in a tool) for the vision and/or goals, where every change is documented with a date and the rationale.
- ▶ A photo of the vision board (or other representation), including notes that reflect the change.

This documentation should be considered as the common memory of the vision and/or goals and is useful to reflect on changes and to recognize the frequency of changes. This frequency is an important metric: too frequent changes, especially in later stages of product development, should be considered an indicator that the overall product development is in danger since there is no clear overall direction for the product.

2.1.4 Specifying the System Boundary Fundamentals

The concept “System Boundary” consists of a set of terms that allows for precise thinking and documentation of the scope and context of the system. A proper understanding of the term “System Boundary” requires an understanding of the terms “Scope” and “Context”.

The following definitions are included in the IREB glossary:

- ▶ System Boundary: The boundary between a system and its surrounding (system) Context.
- ▶ Context: The part of a system’s environment being relevant for understanding the system and its requirements.
- ▶ Scope: The range of things that can be shaped and designed when developing a system.

Sometimes the Context of a system must remain unchanged and the System Boundary is non-negotiable.

Typical examples are:

- ▶ Replacement of a technical component inside a larger existing system. For example, a software component of an embedded control unit in an existing car production line must be replaced due to changed legal requirements. The cars are already in use and the component must fit within the existing interfaces and hardware. Changing these aspects is not possible.
- ▶ Development of a system within an existing ecosystem. For example, an insurance company has a web portal for customers to manage their insurance contracts. The company has decided to develop a smartphone app as a second channel for customers. The app will have the same functionality as the web portal. The app development project may not change the portal or the interfaces to other systems.

In many development efforts, however, the scope and the system boundary may be negotiated. That is, elements of the context may indeed be modified during the development effort. This statement may appear abstract and theoretical, but it has a significant impact on every development effort. It must be clear from the beginning which elements of the system context can be modified and which elements must remain unchanged.

A typical situation is the improvement of a business process with a new system. For example, a bank wants to replace a paper-based application for new accounts with a web portal solution². In the existing process, the potential customer sends the paper application form to the bank. A bank clerk approves the paper application and sets up the bank account by entering the data into the banking system. The new system provides a web-based application for potential customers: the customer enters his or her data into the form and sends the data to the bank. Immediately after sending the data, the customer receives confirmation of the application via email. This is the intended modification in the system context (potential customers no longer use a paper form, they now use the web-based application).

The more interesting part of this example is the process in the back office. Here, three scenarios could be possible:

1. The application data is sent via email to the bank clerk. The bank clerk performs the existing approval process and enters the data manually into the banking system.
2. The approval process is performed within the new web portal by the bank clerk. The bank clerk checks the application data within the web portal. If the clerk accepts the application, then the web portal uses a new interface to the banking system to setup the bank account automatically.
3. The approval process is performed automatically by the new web portal. The web portal is equipped with a rule-based approval engine that allows automatic approval of standard applications. In case the application is approved, the web portal sets up the bank account automatically. In case it's not approved, the application must be checked by a bank clerk.

This is of course an over-simplified and incomplete example, but it shows the impact of the scope decision. In the first scenario, the scope is limited to the web portal, the new application process, and the email transfer of application data to the clerk. In the second scenario, the scope also includes the way the bank clerk works in the back office and the data transfer to the banking system, but the decision on the application remains with the clerk. In the third scenario, even the decision process has become part of the scope of the project.

Which of the three scenarios is appropriate for the bank concerned is not clear from our example and depends on various factors that must be identified and decided during the development effort.

² *Comment:* The description of this example is incomplete by intention. We will uncover further missing aspects on the following pages to illustrate the benefit of various techniques. In case you believe that you already have spotted some missing things, note them and see if we share your viewpoint.

Nevertheless, the bank example shows that a shared and common understanding of the scope and the context of the system is a prerequisite for an effective and efficient development effort. Misunderstandings related to the system boundary or the scope may lead to:

- ▶ Development of functionalities or components that were not under the responsibility of the development effort. For example, our bank project has started to develop the rule-based approval engine (scenario 3), but the stakeholders never agreed on such an approval engine. If the stakeholders decide that this approval engine is not required, then the development effort for this engine is lost.
- ▶ The wrong assumption that functionalities or components that are in fact part of the system should have been developed outside the system (the assumed scope was too small). For example, our example bank project has implemented the email transfer of application data to the clerk (scenario 1), but the stakeholder expected that the approval has to be performed inside the web portal (scenario 2).

The system and the context boundary can be defined by discussing:

- a) *Which features or functionalities have to be provided by the system and which have to be provided by the context?* This question targets the system boundary by talking about concrete capabilities of the system. For example, in our bank project, may the system approve an application automatically or not (scenario 2 or 3)? Another discussion could be the setup of a new bank account: should the new system perform this task or not (scenario 1 or 2)?
- b) *Which technical or user interfaces have to be provided by the system to the context?* This question targets the system boundary and is closely related to the feature/functionality question above. Many functionalities require interfaces to users or other systems. For example, in our bank project, the automatic setup of new bank accounts (scenario 2) requires an interface to the banking system. Also, the approval by the bank clerk inside the web portal (scenario 2) requires a user interface to display and approve the application data.
- c) *Which aspects of the context are relevant / irrelevant for the system?* This question targets the context boundary by explicitly addressing aspects of the context that have to be examined during system development. For example, in our bank project, the application form and the process for sending the data to the bank is definitely part of the context. Whereas the setup of the bank account may be part of the context (scenario 2 and 3) or may be outside of the context (scenario 1).
- d) *Which aspects of the system context can be modified during system development?* This question targets the scope of the system by defining which context aspects may be modified or not. It is important to recognize that an element in the scope is per definition part of the system context. For example, in our bank project, it could be the case that the approval decision must remain with the bank clerk (making scenario 3 impossible).

All four questions are of course closely related and must be discussed together. Keep in mind that the Context Boundary is always incomplete as it can only be defined by the things that one explicitly excludes from the System Context. Similarly, the Scope is never final and may change during a development effort. The important message from a Requirements Engineering perspective is that changes in Scope, System Boundary and Context Boundary must be made explicit for all relevant stakeholders.

2.1.5 Documentation of the System Boundary

The scope and the system boundary can be documented and clarified with several techniques. In this chapter, we will present four of these: context diagrams, natural language, use case diagrams and story maps.

2.1.5.1 Context Diagram

The context diagram is an element of the essential system analysis [McPa1984] and uses diagrams to represent the context. It documents the system, aspects of the context, and their relationship. The notation of a context diagram consists of three modeling elements:

- The system under consideration (circle);
- Aspects of the context (boxes);
- Arrows to represent connections between elements. The direction of the arrow represents the flow of information.

The following figure shows the context diagrams for all three scenarios of the bank account application portal.

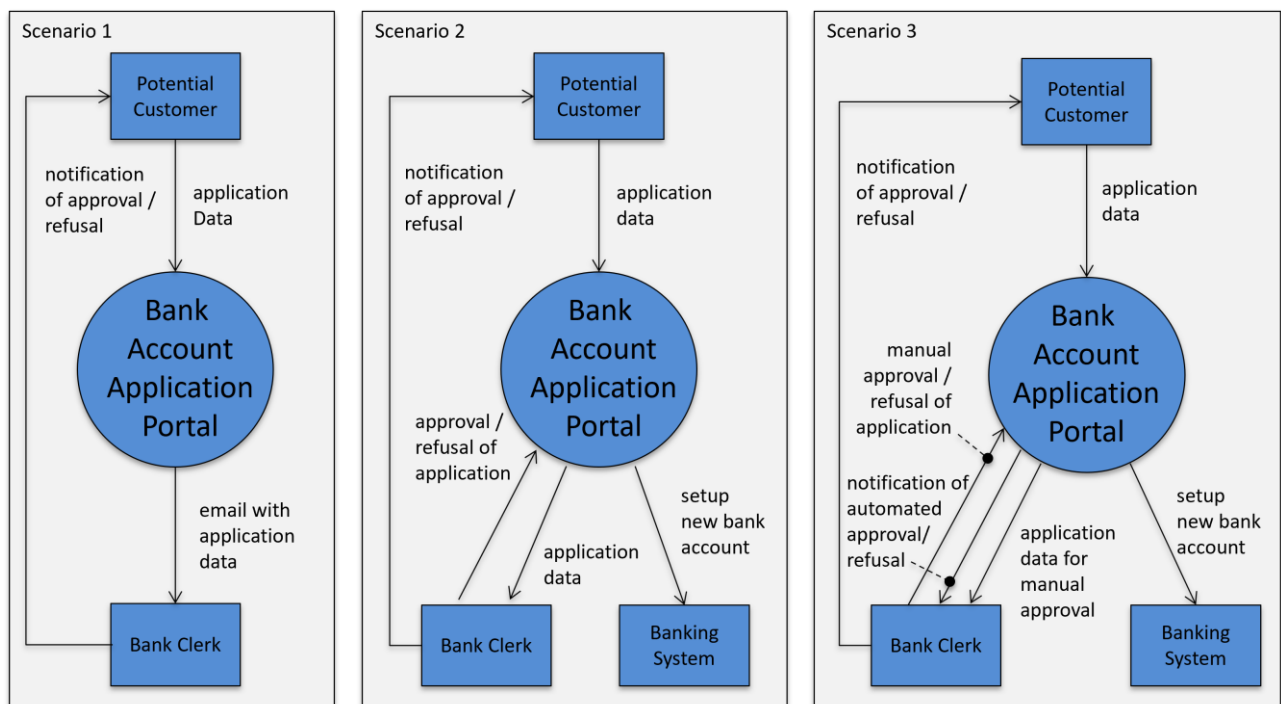


Figure 2: Three context diagrams for the bank account application portal example

All three scenarios include the relationship between the potential customer and the portal (the customer sends application data to the portal) and the relationship between the bank clerk and the potential customer (the bank clerk sends a notification for approval/refusal of the application to the potential customer). Documenting the second relationship (between the potential customer and the bank clerk) is not part of the original context diagram. However, it is useful to document this relationship in practice, since it clearly denotes that the system is not responsible for sending this notification.

The context diagrams for scenarios 2 and 3 share the relationship with the banking system to setup the new bank account in case of approval. This relationship is not part of scenario 1, since the bank clerk sets up the account manually.

One could argue that the relationship between the bank clerk and the banking system could also be documented in the context diagram for scenario 1. There are arguments for and against this:

- For: Setting up the bank account is part of the overall business process (create a bank account) and it must be documented to understand the overall context.
- Against: Setting up the bank account has been defined as out-of-scope for scenario 1. Therefore, it should not be documented.
- Both arguments are understandable and valid. The decision for or against the documentation of such relationships depends on the overall project context.

The main difference between all three scenarios is the relationship between the bank clerk and the portal. In scenario 2, the bank clerk receives all application data and must approve or refuse them. In scenario 3, the bank clerk only receives those applications that cannot be decided upon automatically. In addition, the context diagram for scenario 3 reveals a new and previously missing aspect: the bank clerk receives a notification for automatically approved/refused applications. This information is necessary for the bank clerk to send a notification to the potential customer.

Although the portal example is an oversimplified example, the context diagrams for all three scenarios are substantially different and provide an easy overview of the system and the context.

2.1.5.2 Natural Language Documentation of Scope and System Boundary

Natural Language is the most flexible and easy to use technique to document Scope and System Context. Just provide a list of features/functionalities of the system and a list of further aspects to document the System Context (remember to document aspects that are considered outside as well). Use an additional list to document the Scope of the system.

The Scope and System Boundary documentation from scenario 1 of the banking project could be represented by the following list.

Scope and System Boundary of the Bank Account Application Portal (Scenario 1)

Features/functionalities of the system:

- Web-based form to apply for a bank account
- Send email to customer to confirm having received the application form
- Send application data via email to the bank back office

Aspects inside the context:

- Customer who wants to apply for a bank account

Aspects inside the scope:

- Process of filling out the application form (performed by customer)
- Process of sending application data to the bank clerk

Aspects outside the context:

- Bank clerk from the bank back office who approves the application (or not)
- Setup of the bank account (if application is approved)
- Send approval of application to customer (if application is approved)
- Send refusal of application to customer (if application is not approved)

Comparing this list with the description of scenario 1 in chapter 2.2.1 reveals one new aspect that has not been mentioned before: The description does not mention approval or refusal information. It is not clear how the customer is notified of the approved or refused application.

The above list shows that this issue is not part of the context, hence the development effort does not need to concern itself with this topic. Without this explicit statement, it is very likely that different stakeholders might have different expectations on how approval or refusal would be handled with the new system.

2.1.5.3 Use Case Diagram

The Use Case Diagram is part of UML. It is a diagram type that models the actors and the use cases of a system. A use case specifies the behavior of a system from a user's (or other external actor's or for example other system's) perspective: every use case describes some functionality that the system must provide for the actors involved in the use case.

The focus of Use Case Diagrams on actors and their related functions on a detailed level is very useful for clarifying Scope and System Context. The following notation elements of Use Case Diagrams are useful for modeling the System Context:

- ▶ System Boundary (box with name of the system)
- ▶ Actor (stick figure with name below or box with name)
- ▶ Use Case (oval with name of use case)
- ▶ Relationship between Use Case and Actor (line)

Use Case Diagrams also provide notation elements to model relationships between use cases (for example extends and include relationships). The notation elements are used to document more detailed relationships among use cases. This level of detail is typically not useful for an initial clarification of the system context. The following figure shows use case diagrams for all three scenarios of the bank account application portal.

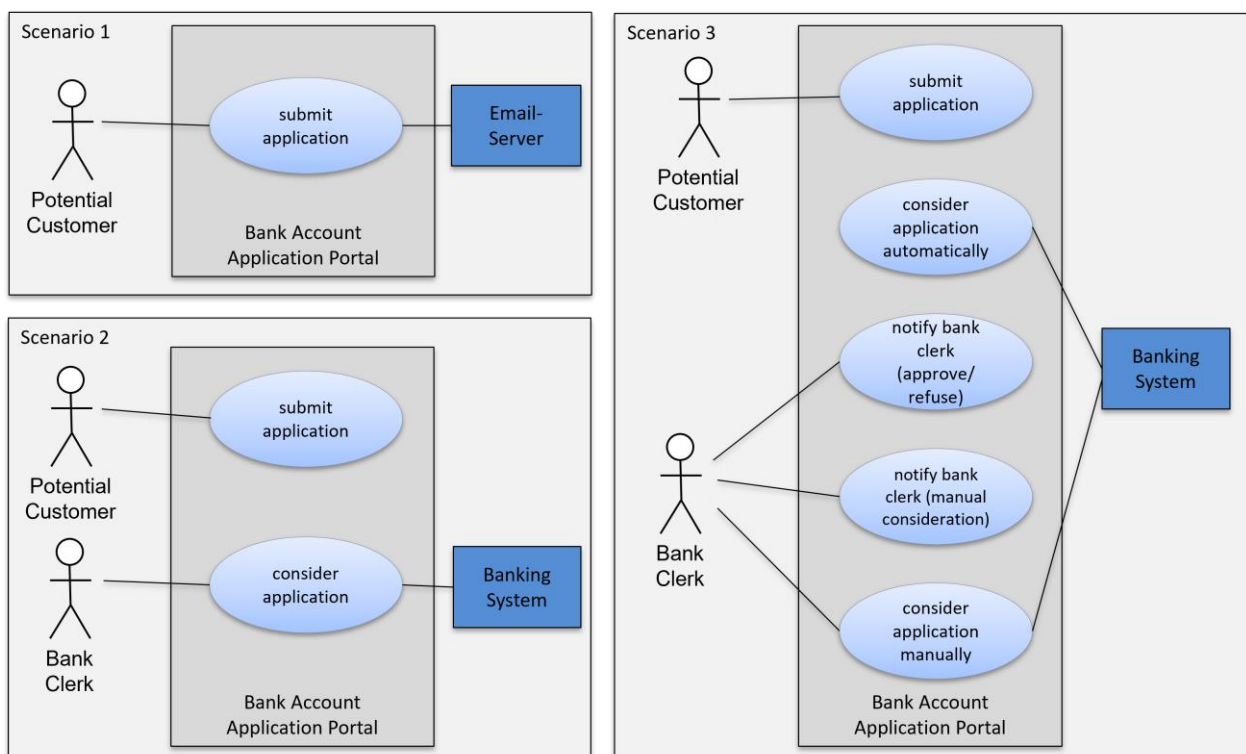


Figure 3: Three use case diagrams for the bank account application portal example

At first sight, the use case diagrams give an overview of the increase of the functional complexity of the three scenarios. Scenario 1 is very simple (one use case), whereas scenario 3 is the most complex one (five use cases).

The core information of the use diagram is carried by the names of the use cases. Therefore, it is important to carefully define proper names for the use cases. Comparing the use case diagrams for scenarios 2 and 3, one can see that the use cases for “considering an application” in scenario 3 are detailed with the adjectives “automatically” and “manually”, to clarify who is performing that task. This clarification is not necessary for scenario 2, since the bank clerk is responsible for considering all applications.

The main differences in terms of system boundary between the three scenarios are clearly visible:

- ▶ In scenario 1, the bank clerk is not part of the system context, since the clerk is not an actor of the portal; the clerk receives the application via email.
- ▶ In scenarios 2 and 3, the bank clerk is part of the system context, since the clerk interacts in various ways with the system.
- ▶ In scenarios 2 and 3, the banking system is an actor, since the portal has to interact with the banking system for setting up bank accounts.

One aspect of the process is not mentioned in the diagrams: the notification of the customer in case of approval or refusal. If this notification is part of the banking system, then the diagrams are correct and the notification is out of scope. But, if this is notification is part of the application portal, the diagrams must be extended to include the notification.

Comparing the use case diagrams and the context diagram (see Figure 2), the main differences between the two notations can be seen:

- ▶ In the context diagram for scenario 1, the bank clerk is documented, since the clerk is an element of the system context that receives information (via email) from the portal. In the use diagram for scenario 1, the bank clerk is not present since the clerk is not an actor with respect to the portal.
- ▶ The use case diagram does not allow documenting relationships between actors that are outside the system, but inside the system context. The context diagram allows documentation of the information flow between context elements (the notification of approval/refusal from clerk to potential customer).
- ▶ The use case diagram provides an initial functional decomposition of the system (the use cases). This functional decomposition is not visible in the context diagram.

These differences originate from the notation elements of both diagrams and should not be understood as advantages or disadvantages of one diagram over the other. If possible, one should create a context diagram and a use case diagram in parallel to benefit from the strengths of both diagrams. If one must choose between context and use case diagram, the following rule of thumb is helpful: if the system under consideration is embedded in a complex context with various important interactions outside the system, then a context diagram would be preferable. If the system under consideration has a complicated set of functionalities and interactions with various users and/or related systems, then a use case diagram would be preferred.

2.1.5.4 Story Map

A Story Map [Patton2014] is a technique for documenting and managing product development during the whole product development process. Its main structure is a two-dimensional arrangement of user stories. The horizontal dimension focuses on the backbone, meaning the narrative flow of the system (or the overall process provided by the system). The vertical dimension provides details for each part of the narrative flow as well as a separation of items according to the development sequence of the software.

Thus, the Story Map provides a useful model for understanding the functionality of the system and describing context/scope on a detailed level. Further details on Story Maps will be presented in chapter 3.4.

2.1.6 The Inevitability of a Changing Scope

The definition of an initial scope (including the system context) must take place at the beginning of a development effort. Without a clear scope, the team has no frame for their development effort and without an understanding of the context, the team has no understanding where the system will be situated and no understanding where to look for information about what to develop.

Nevertheless, scope and system context are never final and stable. In fact, the only event that would make the scope and system context stable would be to take the system out of operation! There are many reasons that require an adjustment of the scope and/or the context. The customer may demand changes and require new functionalities; changes may be necessary as the result of new or modified legislation.

The most common reason, however, for changing the scope and/or the system context, is the evolving understanding of the developers and/or of the stakeholders. In general, every development effort constitutes a significant change in the system context and these changes cannot be fully predicted. Learning new things is natural in such situations, and new learning often has an impact on scope and/or system context.

This situation is not an excuse for not having a proper definition of scope and system context. From a Requirements Engineering perspective, in fact, it is the main reason for defining scope and context systematically. Without a proper initial understanding of current scope and system context, it is only a matter of chance whether the need to adjust it later, will even be recognized. The techniques presented in this chapter are lightweight and easy to use. Using the techniques properly requires only a little effort and provides huge benefits to every development effort.

2.2 Stakeholder Identification and Management

2.2.1 Fundamentals

According to the IREB glossary, a Stakeholder is a person or organization that has a (direct or indirect) influence on the requirements of a system. Furthermore, indirect influence also includes situations where a person or organization is impacted by the system.

This definition emphasizes the importance of Stakeholders, the proper identification of Stakeholders and of Stakeholder Management during the development effort. The statement “responding to change over following a plan” from the Agile Manifesto is often misunderstood and used as an excuse to skip proper Stakeholder Identification at the beginning of a development effort. The identification of a new stakeholder is an inevitable change to which the team must react.

Failure to identify and include an important stakeholder in a development effort can have a major impact: important requirements can be discovered (too) late, or even missed altogether. This may lead to expensive changes late(r) in the process or even a useless system. Stakeholder Identification and Management is an important investment to minimize the risk of missing important stakeholders and their requirements.

2.2.2 Identification of Stakeholders

In this chapter we present the onion model as a simple technique for Stakeholder Identification and classification. Furthermore, the importance of users as central stakeholders, as well as the importance of indirect stakeholders, is discussed.

2.2.2.1 Onion model for stakeholder identification and classification

The Onion Model from Ian Alexander [Alexander2005] is a simple tool for Stakeholder Identification and classification. The model consists of three types of stakeholders (onion layers) that can be systematically searched for stakeholders:

- ▶ Stakeholders of the system: these stakeholders are directly affected by the new or modified system. Typical examples of this class are users, maintenance personnel and system administrators.
- ▶ Stakeholders of the surrounding context: these stakeholders are indirectly affected by the new or modified system. Typical examples of this class are managers of users, project owners, sponsors, or owners of connected systems (for example systems that have an interface with the system under development, see chapter 2.2.4).
- ▶ Stakeholders from the wider context: these stakeholders have an indirect relationship to the new or modified system or to the development project. Typical examples of this class are legislators, standard setting bodies, competitors, non-governmental organizations (NGO's), Trade Unions, Environmental Protection Agencies et cetera.

Stakeholders of the system are also called *direct stakeholders*. Stakeholders from the surrounding and wider context are also called *indirect stakeholders*.

The onion model can be applied in several settings for Stakeholder Identification:

- ▶ Thinking tool - use the onion model to systematically think about the system under development and to write down every possible stakeholder that comes to mind for each layer.
- ▶ Interview guideline - use the onion model as a guideline for short stakeholder interviews. During the interview, the stakeholder can be asked for potential stakeholders within each layer of the onion.
- ▶ Workshop guideline - use the onion model to structure a workshop for stakeholder identification. The model can be used as a visualization tool (for example on a board or flip chart). Each layer of the onion can be analyzed with the workshop participants. For example, every stakeholder writes the names of stakeholders on a card. Alternatively, each layer can be elaborated during a brainstorming session.

As a rule of thumb, the identification of stakeholders should rely on a broad range of sources. A single interview with one person is typically not sufficient to identify the most important stakeholders. Instead, the Product Owner should plan for several interviews and/or workshops for stakeholder identification. If certain names are mentioned several times (for example Maria is referred to as a very knowledgeable person on some business topics), then this redundancy should be interpreted as a sign of importance and not as time wasted.

2.2.2.2 Importance of the user as a direct stakeholder

If a system has human users, these users are amongst the most important direct stakeholders. The success of a system relies on the acceptance of the system by its users. Even if the features of a system are perfectly implemented, then the system is worthless if the users do not want to use the system.

A simple classification with respect to stakeholders is the separation between open and closed environments:

- ▶ In an open environment, the users have alternatives to select from. For example, a company wants to develop new office software (for example for word processing and presentations). There are several alternatives on the market for this kind of product. The stakeholder analysis must focus on information that helps to convince users to switch from their existing system to a new one.
- ▶ In a closed environment, the users are “forced” to use a new system. For example, a company develops a new business administration system for managing their business and every employee of that company must use this new system because it is connected to every part of the company. In such a closed environment, stakeholder identification (and management) may not receive sufficient attention, because the users have no choice but to use the system. Such behavior underestimates the power of the corporate immune system: if the users do not accept the new system, then the immune system of that organization will find a way to prevent its introduction.

The users of a system (in both open and closed environments) typically cover a wide spectrum of people with different expectations, attitudes, and prerequisites. Understanding the spectrum of users for a system is an important first step.

If the number of users is small, it is advisable to get to know them (or their representatives) via personal interviews. In such situations, the users can be asked requirements-related questions directly.

If the number of users is large or even unknown (typically in open environments), the spectrum of users should be captured using other means. A proper tool for such a situation is the Persona Technique [Cooper2004]. A Persona represents an example user with distinct characteristics. Such a Persona is typically described in a detailed way including a real name (for instance Jim), one or more pictures, a short CV and a list of hobbies. The goal of the description is to illustrate the persona as realistic as possible and to ask requirements-related questions to this persona (for instance: What kind of search function would Jim prefer?). A single persona is typically not sufficient for a development effort. As a rule of thumb, a project should develop 3-5 persona with various backgrounds. It is especially advisable to develop persona with distinct positions (for instance a novice and an expert business person). If new software is developed for these selected distinct user profiles representing the extreme usage scenarios of the product, then most mainstream users (for example the average or experienced user) will also accept the system.

Persona is a technique that is embedded in the design process of new software. An alternative, more measurement-oriented approach is the application of data analytics, Google analytics and big data: The behavior of online users can often be analyzed directly by embedding such technologies into deployed product increments. The main benefit of such techniques is that they provide concrete data on user behavior. The main drawback of such techniques is that they must be planned in detail and implemented into the software increment. Hence, the measurement objectives for such techniques have to be clear since gathering of the related data is expensive.

2.2.2.3 Importance of indirect stakeholders

Indirect stakeholders can be found in the surrounding context of the system and may be as important for a development effort as the users themselves. The term indirect stakeholder is by intention very broad since indirect stakeholders differ significantly for different types of systems. The general idea behind indirect stakeholders is to search for stakeholders that can have impact on the success of a system, either positively (support) or negatively.

Support can be provided in various forms. A stakeholder can provide important information related to the domain (for example important business rules or user needs) or on future developments in the domain (for example a new type of product, a new law that may impact the business). A stakeholder can also provide political support during the development and introduction of the system (for example an important manager from a related department).

A negative impact on the success of a system may also happen in various ways. An important aspect may be, for example, the formal admission of a system in regulated environments (for example medical systems): if relevant stakeholders for the admission of a system are not involved early in the development process, then a new system may fail to fulfill important admission criteria. The political dimension of a development effort is another aspect (for example a manager of a department with a competing product may hinder the development). The negative impact is not limited to the development effort. Underestimated types of indirect stakeholders are NGO's or people that are only loosely related to a system. For example, an NGO that is active in the field of personal health data protection may have a strong view on storing certain types of personal health related data. If you develop a system in this area, then such an NGO may start a campaign against your system.

Investing time in the identification of indirect stakeholders should be considered as a means of gathering additional information for the development process in order to reduce the risk of failure. As a rule of thumb, a Product Owner responsible for Requirements Engineering should develop a broad view on indirect stakeholders. Talking to indirect stakeholders is often beneficial, even if an indirect stakeholder does not provide new insights; the confirmation of already known information is often also beneficial.

2.2.3 Management of Stakeholders

Systematic identification of key stakeholders must take place at the beginning of a development effort as a setup activity. Managing the identified stakeholders throughout the development effort is a continuous activity. Although this sounds very costly, a simple list including contact details and relevant attributes (for example areas of competence or availability) will suffice in most contexts. If the project uses a wiki to manage the documentation, then the stakeholder list can be created and maintained easily in the wiki.

The list may change at any time, either because a stakeholder was initially overlooked or due to changes in the context, such as a new NGO being established. Once a new stakeholder has been identified, the stakeholder should be approached to elicit the requirements for the new system and to gather other valuable information.

Because of the broad range of possible stakeholders, every participant in a development effort (for example the developers and the Product Owner) should participate in the identification of missing stakeholders. The first step is to create awareness among the developers about the importance of stakeholders and to look for signs of new or missing stakeholders.

2.2.4 Sources for Requirements beyond Stakeholder

Depending on the system and the domain, existing documentation, neighboring systems with interfaces to the developed system, legacy systems or even competitor systems may also be important sources of requirements. The following list provides some examples:

- ▶ If the system under development has a predecessor system, the documentation (if there is any) and the source code of this legacy system can provide important requirements (for instance detailed requirements on data structures);
- ▶ If the system under development has interfaces to other existing systems (for example in a large business context), the documentation of the interfaces provides important requirements for the interaction between the system under development and these systems. The users, developers et cetera of these existing systems are of course important stakeholders;
- ▶ Almost every system has one or more similar systems, meaning systems that perform similar tasks in other contexts. Such similar systems are often underestimated as a source for requirements and ideas. If you develop, for example, a shopping system for a highly specialized product, then you should have a look at existing online shops and their functionalities to see if they could also be useful for your systems;
- ▶ If developing a highly innovative system, recent research in this area could also be an important source for requirements. There are several Internet databases that can be searched for research material (for instance Google scholar).

If your development effort can benefit from additional sources for requirements, these should be systematically identified and managed in a way similar to managing stakeholders. Detailed information on the management of other requirements sources is provided in the IREB Advanced Level module Elicitation.

2.3 Summary

The definition of Vision and Goals, Stakeholders, System Scope and Context are interdependent:

- ▶ Relevant stakeholders formulate the vision and the goals. Therefore, the identification of a new stakeholder may have an impact on the vision or the goals.
- ▶ The vision and goals can be used to guide the identification of new stakeholders by asking: Which stakeholder may be interested in achieving the vision and/or the goals or is affected by achieving the vision and/or the goals?
- ▶ Vision and goals can be used to define an initial scope by asking: which elements are necessary to achieve the vision and/or the goals?
- ▶ Changing the system boundary (and thus the scope) may have an impact on the vision and/or the goals. If aspects are removed from the scope, then the system may no longer have sufficient means to achieve the vision and/or the goals. Conversely, if the scope is extended, this may provide new means to fulfill the vision and/or the goals.

- ▶ Stakeholders suggest the system scope. Therefore, the identification of a new, relevant stakeholder may have an impact on the scope. For example, an important manager may decide that the scope of the project can be extended.
- ▶ A change of the scope (for example to fulfill a new or modified goal) requires agreement from the relevant stakeholders.

These strong interdependencies mean that it is important to balance all three elements and to examine the impact of changing one of the three elements on the others. Being aware of these interdependencies is the first step towards working jointly on vision and goals, stakeholders and scope. Because of these tight interdependencies, we recommend handling these elements together.

Before starting with iterative development, we recommend creating a coherent, initial specification that includes:

- ▶ vision and/or goals
- ▶ scope and system context
- ▶ initial list of stakeholders (and potentially other sources)

The methods and tools presented in this chapter can be used in a lightweight way to create such a specification. A good, lightweight starting point is a half-day workshop with all three elements on the agenda. Every participant should prepare for the workshop by answering the following questions:

- ▶ What is your vision for the system? What are the most important goals for you?
- ▶ What is your understanding of the system context and the scope?
- ▶ Which stakeholders and other sources (documents, systems) have to be considered for the project?

If the workshop participants are not familiar with the terminology, provide the definitions as background information to them. The outcome of this workshop is a starting point for the creation of an initial specification using the methods and/or techniques described in this chapter.

The initial specification should be considered as a living document that should be checked and updated on a regular basis. The rituals and techniques of agile development provide several ways for lightweight maintenance of this documentation. A pragmatic approach is to include a crosscheck against context/scope documentation in the definition of ready. For example, if the scope was described by means of a use case diagram, then every user story would be linked to a use case and actor.

2.4 Case Study and Exercises

Throughout this Handbook we will use a case study. Imagine that you want to create a system that allows students to use a training platform via Internet to learn about Requirements Engineering. Short video lessons should be offered together with questions to assess whether a student mastered the various topics. The platform should be useable on any device that allows the students to connect to the Internet, meaning smart phones, tablets, laptops, ... For the manager of a larger group of students the platform should offer information about the progress of the individual students. We suggest calling the platform "iLearnRE".

Suggestions for Exercises:

If you want to practice the Clean Project Start, we invite you to use the iLearnRE case study. As an initial list for the vision and/or goals, we have defined the following statements:

- Online Video Training Portal to learn about Requirements Engineering and prepare for the IREB exam
- Available on different platforms even with low-bandwidth internet connections
- Includes a chat room/discussion forum to discuss issues with other students
- A management dashboard to control progress of students in your team

We have further defined the following list of users:

- Students
- Administrator of the Portal
- Team Leaders (of Students)
- Question Authors

With this information, you can work on the following exercise:

- 1) Use the techniques from paragraph 2.1.2 to reformulate the vision/goal statements
- 2) Create a context diagram for iLearnRE
- 3) Create a use case diagram for iLearnRE
- 4) Think about additional stakeholders for iLearnRE

3. Handling Functional Requirements

In chapter 2 you have learned about the clean project start, for instance about important prerequisites that you should gather before beginning iterative, incremental development.

This chapter deals with eliciting, discussing and capturing Functional Requirements. The other two categories of requirements (quality requirements and constraints) will be discussed in chapter 4. Many of the ideas in this chapter are also relevant for these other two categories.

In this chapter you will learn that it is quite normal that stakeholders talk on different levels of granularity all the time. They will sometimes ask for very abstract things, where you as a Product Owner will have to work quite hard to find out all relevant details. And sometimes they will ask for very small, precise things that are already close to what developers can understand and implement. Your job as a Product Owner is to deal with all these levels of granularity. High level is not bad if these features are not needed very soon. But for those that should be implemented in one of the next iterations more precision is required.

In the agile world, coarse-grained requirements are often called epics, themes or features. This chapter will show you how to transform them into INVEST user stories, that is to make them precise enough so that they can be dealt with by the developers.

As soon as you accept the idea that requirements do exist on different levels of granularity, some questions naturally arise:

- ▶ How do we deal with multiple levels of granularity?
- ▶ Which criteria can and should be applied to split big, abstract topics into smaller chunks?
- ▶ Is it sometimes necessary to group many small requirements into larger chunks so that we have a “bigger picture” for orientation?
- ▶ How precise do we have to be before the developers can begin with the implementation?
- ▶ Is it necessary or advisable to keep multiple levels of requirements, or can we throw away abstract statements as soon as we have more concrete requirements?
- ▶ Do we have to capture all of this in writing or can we simply talk about it?

In this chapter we will deal with all those questions. As mentioned earlier we will concentrate on Functional Requirements. In chapter 4 we will discuss quality requirements and constraints. In chapter 5 Estimation, Ordering and Prioritizing of Requirements will be discussed. This chapter 3 is solely about managing complex functional requirements and refining them to a level such that they can be taken on by developers.

3.1 Different Levels of Requirements Granularity

Let us take some examples from our case study “iLearnRE“. We can formulate one of our goals as: “As a student I want to learn about Requirements Engineering in an online video course, so I do not have to go to a classroom“.

Let us assume that one of your stakeholders now asks for the following feature:

“As a department head I want to be able to check the learning progress of all my employees“

This is not a very precise statement, since we do not necessarily understand what “progress” means. Also, we do not know what the result of this check should look like. But it is a relevant request. We would characterize this as a coarse-grained requirement.

Assume that one of the students comes up with the request:

“While playing a video clip I want to be able to see the rest of its runtime in seconds”

This is a more precise requirement that still needs some more details for implementation (location, size, color of the runtime display) for implementation. These details can be added by the Product Owner, which will lead to a solution by the Product Owner, which is not necessarily the best solution. Or the Product Owner asks the team during the refinement meeting for options regarding the details and decides based on the available information.

Stakeholders constantly talk to us on all levels of granularity. As a Product Owner you cannot, and should not, force them to be more structured. Working with these different levels of requirements and structuring them is your job as Product Owner, with the support of those helping you during the Requirements Engineering process.

As Figure 4 shows, every system will have requirements on different levels of granularity below the top-level vision and/or goals. As Product Owner you are striving for two goals:

1. To have an overview of all currently known functional requirements. This allows to select the most valuable ones for early implementation, to keep the bigger picture in mind et cetera;
2. Understand requirements in enough detail so that they can be taken on by the developers for implementation.

Some methods give specific names to the levels of requirements. SAFe for example calls the big chunks “epics”, the mid-size requirements “features”³ and the lower level requirements “user stories”. Other popular names for more abstract requirements are “topics” or “themes”.

There is no consensus in the Agile community about the terminology for more abstract requirements. We will discuss these terms in chapters 3.2 and 3.6.

During the requirements elicitation and documentation process, this *hierarchy of granularity* can be established in different ways. As mentioned earlier, stakeholders typically tell you their wishes at various levels. So you can try working top-down (from visions and/or goals to lower level requirements), bottom-up (grouping lower level requirements into larger chunks), or middle-out (starting with requirements in the middle, breaking some down into more detail while others are grouped together).

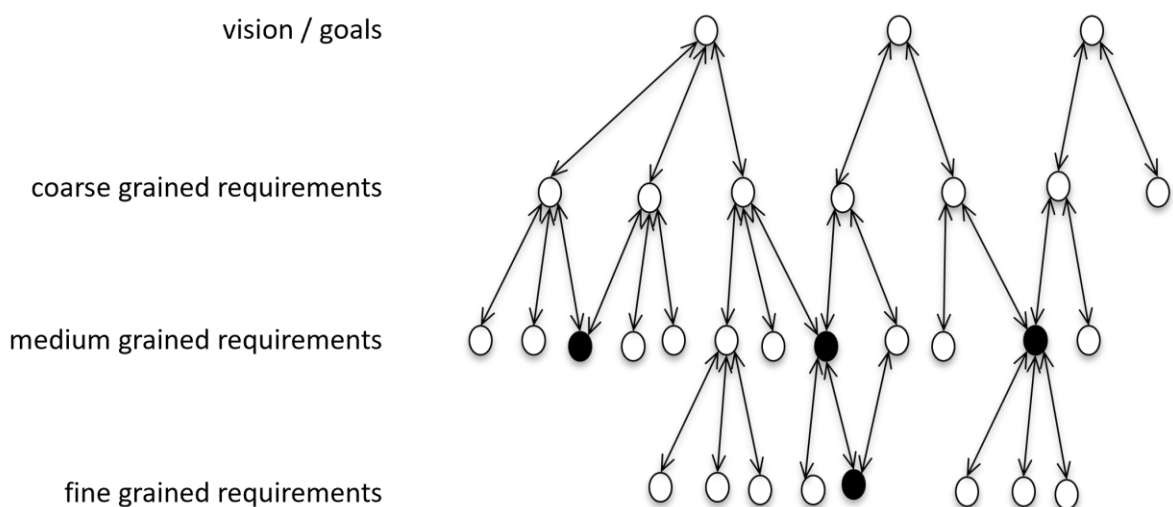


Figure 4: Requirements on different levels of granularity

³ SAFe has also the optional Level “Capabilities” between Epics and Features.

As Product Owner you should maintain the relationships (traces or links) between all requirements. This will not only give you a better overview, but will also allow you to discard requirements that are not goal-oriented. Thus, you can avoid requirements creep and concentrate on those that should really be achieved.

Note that some detailed requirements can be part of multiple, higher-level requirements, as indicated by the black dots in Figure 4, for instance one detailed activity may be performed as part of two or more business processes.

Figure 5 shows some example requirements from the case study iLearnRE, including their links.

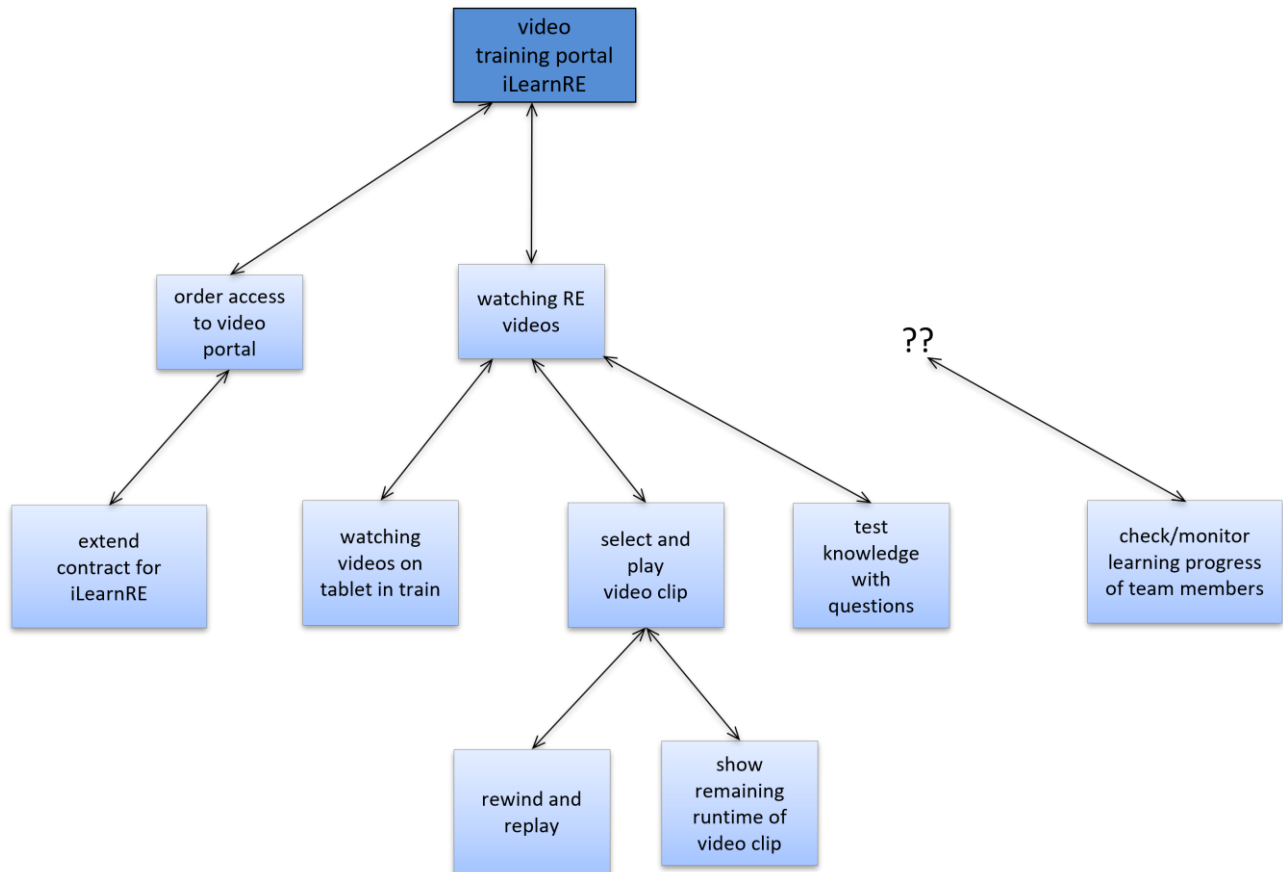


Figure 5: Sample requirements from the case study

Such a structured hierarchy of requirements will allow the Product Owner (and all other stakeholders) to avoid the risk of being lost in a larger project. The levels in this hierarchy can be used to come up with estimates and they can be used to prioritize requirements. This will be discussed in more detail in chapter 5.

Criteria for grouping or splitting requirements, useful notations to capture them, and tools and techniques to support the overview will be discussed in the next chapters.

3.2 Communicating and Documenting on Different Levels

The vision and/or the goals have to be made more precise in order to come up with functional requirements that can be communicated to and implemented by the developers.

Based on the principle of “divide and conquer“, we need to decompose a large system or product into smaller parts. Figure 6 illustrates this approach. We will discuss strategies and tactics how to achieve this goal.

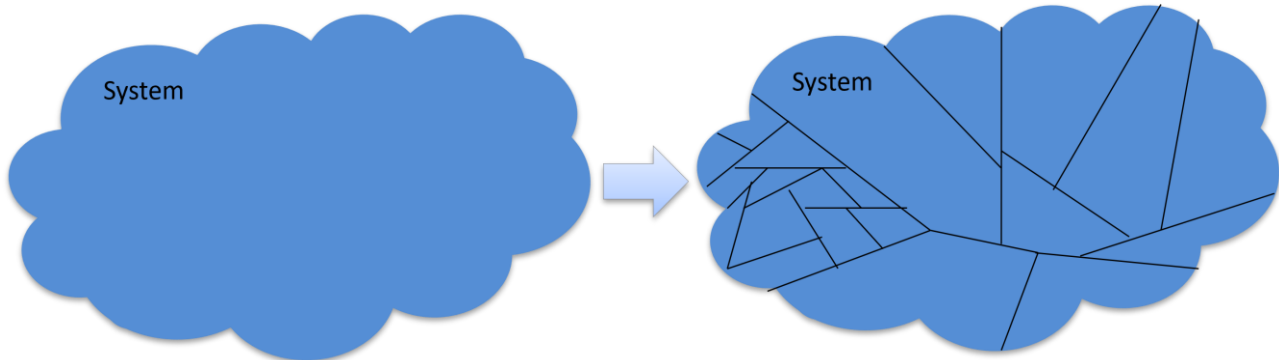


Figure 6: Decomposing functional requirements

Here are some approaches for decomposing a large system (including examples from the iLearnRE case study):

1. Split into logical functions (also called features, epics or themes):
For example: Establishing a contract for e-learning, watching videos, testing your knowledge with questions or checking learning progress
2. Use history, for instance the structure of an existing product, as a partitioning theme:
Since we have no predecessor project of our case study this strategy does not work here.
3. Split by organizational aspects (meaning parts serving different departments or user groups):
For example: Software for students, software supporting the team leader, software for the admins of the iLearnRE product
4. Split according to hardware:
For examples: iLearnRE desktop with responsive design, iPhone native app, Android native app
5. Split by geographical distribution:
For example: iLearnRE for a country with the highest number of potential users, extension to other countries with different legislation.
6. Split by data (business objects):
For example: functions dealing with videos, functions around questions, functions around contracts and functions around invoices
7. Split into externally triggered, value-creating processes.

All of these approaches will result in smaller chunks that can then be analyzed separately.

The first six approaches look at the system's *internal* structures: its functions, its historical structure, its organizational split, its hardware or geographical distribution or its business objects.

Only the last approach (value-creating processes) starts in the context, outside the scope of our system. It looks at external triggers to which our system should react.

These triggers could have different sources: human users needing something from the system, other software systems sending input and requesting some system action, hardware devices (like sensors) triggering an action inside our system.

The context diagram is a valuable source when identifying such external triggers, since it shows all adjacent systems that might request some action from the system under consideration.

This value-oriented decomposition has been suggested by many authors over the past decades: [McPa1984] called it “event-oriented decomposition”, [Jacobson1992] called it “use case decomposition”, [HaCh1993] called it “business processes”, and finally [Cohn2004] called it “user stories”. All of them suggest different notations to capture the results of this decomposition. Figure 7 shows such a decomposition in two of these notations: use cases and user stories.

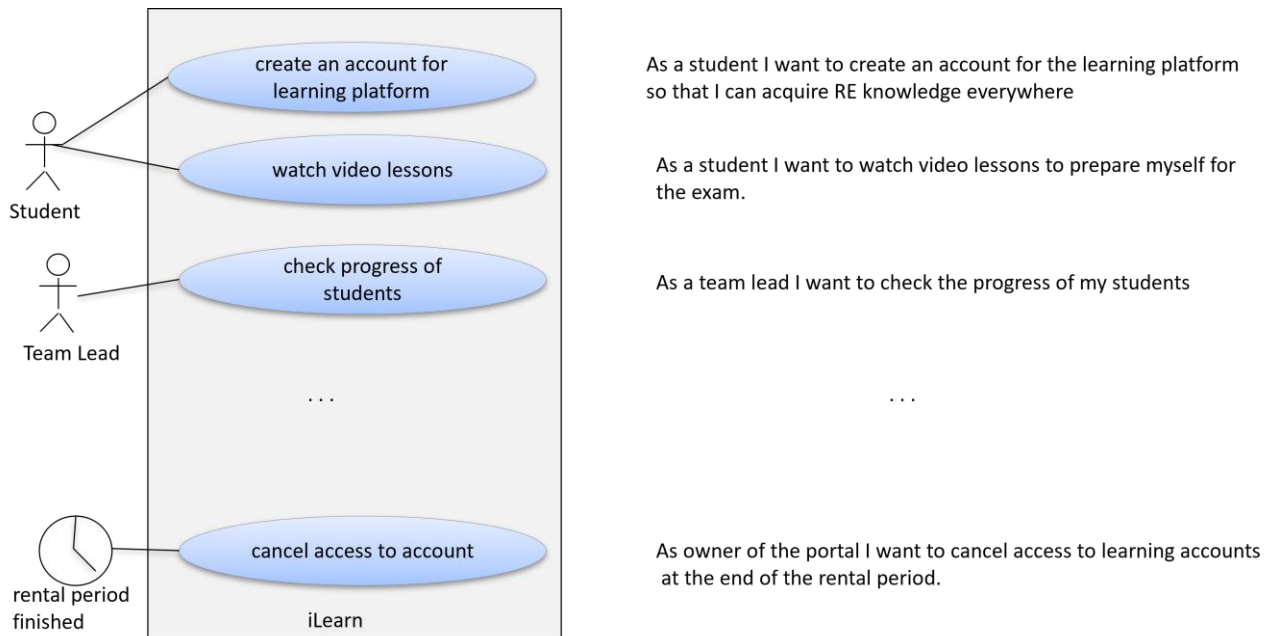


Figure 7: A value-oriented system decomposition into processes in different notations

Let us ignore notations for a moment and study the characteristics of these decompositions. Agile experts will recognize these criteria as the first three of Bill Wakes’ INVEST criteria [Wake2003]).

All the resulting processes are:

I: independent⁴ of each other, meaning they are self-contained and minimize mutual dependencies. They should not overlap in concept, and we would like to be able to schedule and implement them in any order.

N: negotiable, meaning they do not yet represent a fixed contract, but leave space for discussions of the details.

V: valuable: they bring real value to the requester, that is to a person or another system in the context.

The other criteria of INVEST will be discussed in the next chapter about user stories.

The approaches for decomposition as mentioned earlier can also be used. Especially when writing requirements for an existing system, its current structure of components or subsystems is often a good starting point for eliciting new requirements. There is, however, a danger of specification gaps or overlap between those parts (see Figure 8). Since all backlog entries will be discussed and negotiated you would probably catch such gaps and overlaps. But thinking in terms of value creating processes (with whatever notation) avoids these dangers right from the beginning.

⁴ Another interpretation of the letter “I” is “Immediately actionable” [S@S Guide]

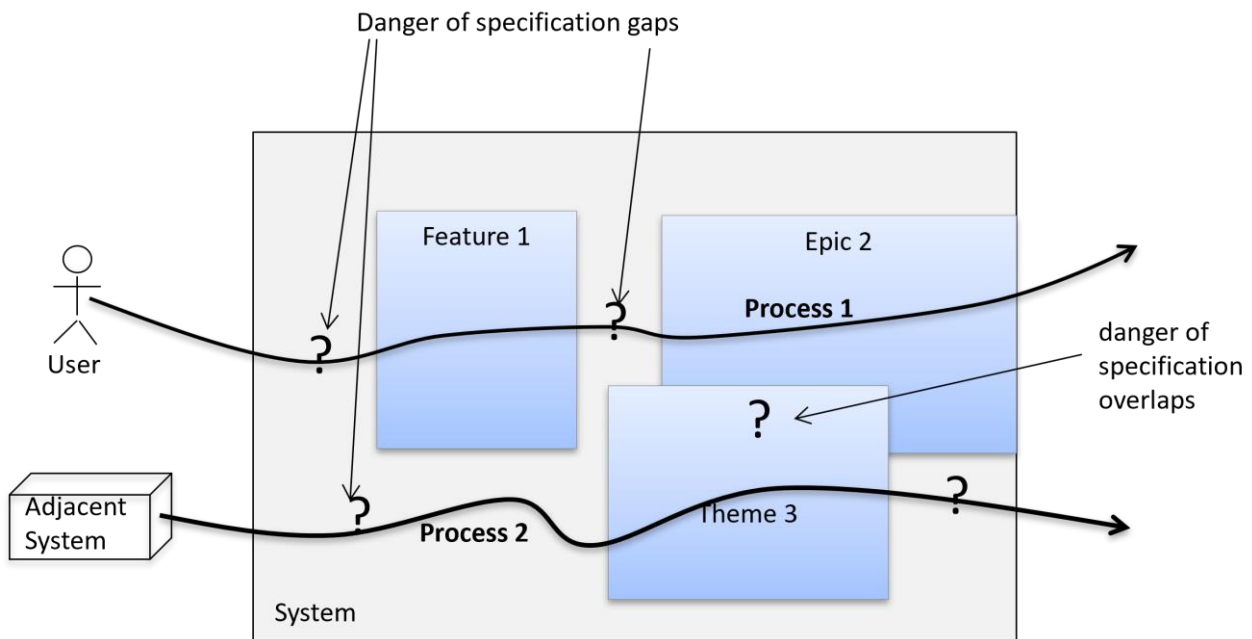


Figure 8: Specification gaps and overlaps

A suggestion about how to come up with a good value-oriented process decomposition is not to think in terms of users or actors of the system, but to identify events that happen in the context and to which the system has to react. [McPa1984] identified two basic kinds of events:

- ▶ External events: Triggered by users or adjacent systems;
- ▶ Temporal events: Triggered by time or observation of system internal resources.

As a Product Owner you might miss the second category since they have no explicit actor or user. The system executes a predefined process without external triggering input, just triggered by time or observation of internal resources.

Examples for both kinds from our case study:

- ▶ External event: “As a student I want to assess my knowledge with test questions.”
- ▶ Temporal event: “Two weeks before the end of the subscription period it is time to remind students about a possible prolongation.”

We have now seen several approaches to find functional requirements to fulfill our visions and goals. The suggestion is to apply a process-oriented decomposition strategy since it helps to identify **I**ndependent, **N**egotiable and **V**aluable chunks of functionality. Any other decomposition strategy that results in such INV-chunks is also fine.

As a Product Owner you want to achieve an overview of the system’s functionality. Of course, your backlog is always open to accept more functionality; however, for decisions about the project roadmap, for rough estimations, or for discussing minimum viable or minimum marketable products, the overview will help you. It is a good basis for deciding where to look for more detail early on.

Having discussed ways to come up with a rough decomposition, let us now concentrate on communicating and documenting these functional requirements.

The basic choice is between drawing and writing. You can visualize a level 1 decomposition of your goals or visions either by drawing a use case diagram, or you can write larger user stories and put them onto separate cards. Figure 7 showed excerpts from our case study in both styles, side-by-side. The following chapter will discuss user stories in more detail.

Note that in principle both notations contain the same amount of information and are equally detailed or abstract. It is more or less a matter of personal taste whether you prefer overview pictures or written stories.

3.3 Working with User Stories

For a Product Owner, user stories are an excellent way to communicate requirements to all stakeholders and also to the developers. User stories are usually captured on story cards, although a multitude of tools is available to capture them electronically. In this chapter we will focus on the idea of user stories.

3.3.1 The 3 C model

As mentioned earlier stories are often written on index cards or sticky notes and arranged on walls or tables to facilitate planning and discussion. This strongly shifts the focus from writing about features to discussing them. In fact, these discussions can be more important than the actual text written on the card or note.

Ron Jeffries [Jeffries2001] summarized this aspect in his 3C-model (Card, Conversation, Confirmation) to distinguish the more social character of stories from the more documentary character of other requirement notations. His ideas are explained in the following chapters:

The “**card**” (an index card or a sticky note) is a physical token, giving tangible and durable form to what would otherwise only be an abstraction. The card does not contain all the information that makes up the requirement. Instead, the card has just enough text to identify the requirement, and to remind everyone what the story is. The card is a token representing the requirement. It is used in planning. Notes are written on it, reflecting for example priority and cost. It's often handed to the programmers when the story is scheduled to be implemented and given back to the customer when the story is complete.

The “**conversation**” takes place at different moments and places during a project, particularly when the story is estimated (usually during release planning) and again at the iteration planning meeting when the story is scheduled for implementation. It involves various people concerned with a given feature of a software product: customers, users, developers, testers - and is largely a verbal exchange of thoughts, opinions and feelings.

The conversation can be supplemented by other requirements, artifacts and documentation. Good supplements are examples; the best examples are executable test cases.

The “**confirmation**”: No matter how much discussion or how much documentation we produce, we cannot be as certain as we need to be about what is to be done. The third C in the user story's key aspects provides the confirmation that we have to have: the acceptance tests.

The confirmation provided by the acceptance test allows us to use the simple approach of card and conversation. When the conversation about a card gets down to the details of the acceptance test, the Product Owner and the developers settle the final details of what needs to be done. When the iteration ends and the implementation team demonstrates the acceptance tests running, the Product Owner learns that the team can, and will, deliver what's needed.

3.3.2 A template for user stories:

Mike Cohn defines user stories in the following way:
(<https://www.mountaingoatsoftware.com/agile/user-stories>):

“User stories are short, simple descriptions of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system. They typically follow a simple template:

As a <type of user> I want <some goal> so that <some reason>.”

Note the three components of this formula. They ensure that:

1. we have someone **who** wants that functionality (“As a user ...”),
2. we know **what** the user wants (“... I want ...”) and
3. we understand the **why**, i.e. the reason or motivation (“... so that”).

The formula helps us to think about Who wants What and Why. It is not so much the formalism that makes user stories successful, it is asking and answering these three questions.

You have seen some examples for stories from our case study iLearn in Figure 7. Here are some additional examples:

- ▶ As a student I want to put questions into a forum so that others can provide answers or opinions.
- ▶ As a questions author I want to add new questions and answers to the pool so that students can test their knowledge.
- ▶ As manager of the portal I want to upload new versions of official IREB questions so that our portal is always up-to-date with IREB.

In his definition Mike Cohn explained that user stories are told from the perspective of the person who desires the new capability. Note that sometimes the word “user” is a bit misleading, since the person wanting a feature is not necessarily the one working with the system as a user.

For instance, in the last example: the administrator who has to upload new versions of official IREB questions is not necessarily the one who wants this to be done. It is the business owner who wants this to be done.

This is especially true for processes that are time-triggered, meaning the process is executed automatically by the system at a particular time or when some condition is fulfilled. A “user” is not needed, but there has to be someone who benefits from the process – otherwise executing the process does not make sense. As a Product Owner or Requirements Engineer you should always search for this beneficiary. Ask yourself: Who really wants this feature and sees value in having the feature?

From a business point of view it often makes sense to talk less about “user stories”, but simply call them “stories” – thus avoiding the explicit reference to a user. But you always have to find out who really wants a specific story. In the following text we will use the short form “stories” as an alias to “user stories” wherever appropriate.

If you want to avoid this discussion, simply refer to everything as a backlog item according to the scrum guide [S@S Guide].

Here is an example from our case study for a process triggered by a temporal event:

- ▶ Two weeks before the end of the subscription period it is time to remind students about a possible prolongation.

If you want to write this feature as a story, according to the template of Mike Cohn, you have to identify the owner of the platform as the beneficiary.

- ▶ As owner of the platform I want an automatic reminder being sent to a student two weeks before the end of the subscription period to give the student the chance to prolong access to the account.

3.3.3 INVEST: Criteria for “good” stories

In 2003 Bill Wake published an article [Wake2003] advising to INVEST in good stories. We have already discussed the first three letters of that acronym in chapter 3.1: Stories should be **I**ndependent of each other, they are **N**egotiable and they must be **V**aluable for someone.

In order to be good enough for implementation by a developer, they also have to fulfill the other three criteria: **E**stimated, **S**mall enough to fit into the next iteration and **T**estable.

Estimation techniques will be discussed in chapter 5.

If the estimate shows that the story is still too big to be implemented in one iteration, it has to be broken up into multiple stories. Splitting techniques for stories are discussed in chapter 3.4.

And, finally, as mentioned above in the chapter about confirmation, stories have to include sufficient details about test cases or acceptance criteria (usually captured on the back side of the card). This represents an agreement on the things that the developers have to demonstrate to the Product Owner at the end of an iteration. See chapter 3.5.

3.3.4 Supplementing stories with other requirements artifacts

As mentioned above, the story on the card does not contain all the information that makes up the requirement. It is just a physical token to foster communication among all stakeholders and team members. Sometimes, it is very useful to use other requirements notations and artifacts to supplement the story on the card.

Feel free to use activity diagrams, BPMN, flow charts or data flow diagrams – in short: everything you have ever used to visualize a business process or a flow of steps.

Example:

To better understand the story “As a student I want to create an account for the learning platform so that I can acquire Requirements Engineering knowledge everywhere”, you could add the following activity diagram:

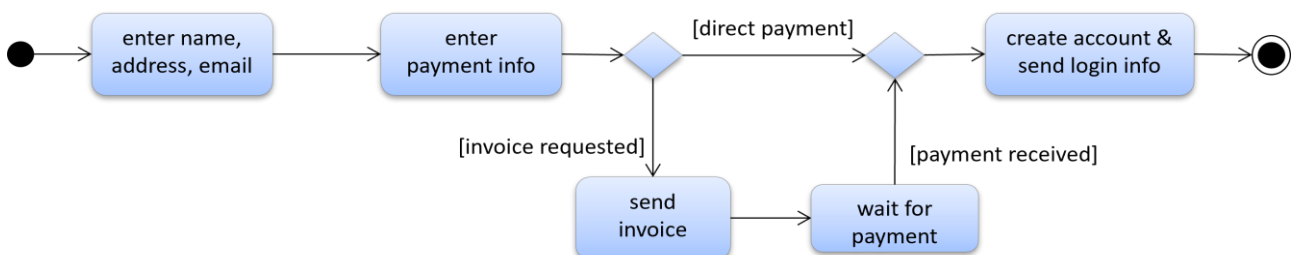


Figure 9: Activity diagram to explain details of a story

3.4 Splitting and Grouping Techniques

In order to generate user stories that are small enough to fit within a single iteration, larger stories may be split into more fine-grained ones. A number of authors have suggested patterns that can be applied for this purpose, ranging from reducing the feature list to narrowing down the business variations or input channels [Leffingwell2010].

One of the most extensive suggestions comes from Lawrence [Lawrence1] and is presented in the form of an easy-to-learn cheat sheet. It advises you to ask yourself the following questions in order to achieve smaller stories:

1. **WORKFLOW:** Does the story describe a workflow? If so, can you split the story in such a way that you do the beginning and the end of the workflow first and enhance with stories from the middle of the workflow later? Or, can you take a thin slice through the workflow first and enhance it with more stories later?
2. **MULTIPLE OPERATIONS:** Does the story include multiple operations? (For instance, is it about “managing” or “configuring” something? Can you split the operations into separate stories?)
3. **BUSINESS RULE VARIATIONS:** Does the story have a variety of business rules? (For instance, is there a domain term in the story like “flexible dates” that suggests several variations?) Can you split the story in such a way that you can do a subset of the rules first and enhance with additional rules later?
4. **VARIATION IN DATA:** Does the story do the same thing to different kinds of data? Can you split the story to process one kind of data first and enhance with the other kinds of data later?
5. **INTERFACE VARIATIONS:** Does the story have a complex interface? Is there a simple version you could do first? Does the story get the same kind of data via multiple interfaces? Can you split the story to handle data from one interface first and enhance it with the others later?
6. **MAJOR EFFORT:** When you apply the obvious split, is whichever story you do first the most difficult? Could you group the later stories and defer the decision about which story comes first?
7. **SIMPLE/COMPLEX:** Does the story have a simple core that provides most of the business value and/or learning? Could you split the story to do that simple core first and enhance it with later stories?
8. **DEFER PERFORMANCE:** Does the story get much of its complexity from satisfying quality requirements like performance? Could you split the story to just make it work first and then enhance it later to satisfy the quality requirements?
9. **Last resort: BREAK OUT A SPIKE:** Are you still baffled about how to split the story? Can you find a small piece you understand well enough to start? If so: Write that story first, build it, and start again at the top of the suggestions. If not, can you define the one to three questions holding you back the most? Write a spike with those questions, do the minimum to answer them, and start again at the top of the suggestions.

Note that even fine-grained user stories should be defined in such a way that they deliver some value for at least one stakeholder. Therefore, slicing a workflow into its individual steps is often counterproductive, since implementing one or the other step may not deliver any value. Therefore [Hruschka2017] suggests rather decomposing a use case (or a large process) into slices that go from end to end. This is based on Ivar Jacobsons idea about use case slices [Jacobson2011]. Figure 10 shows this idea in a graphical format.

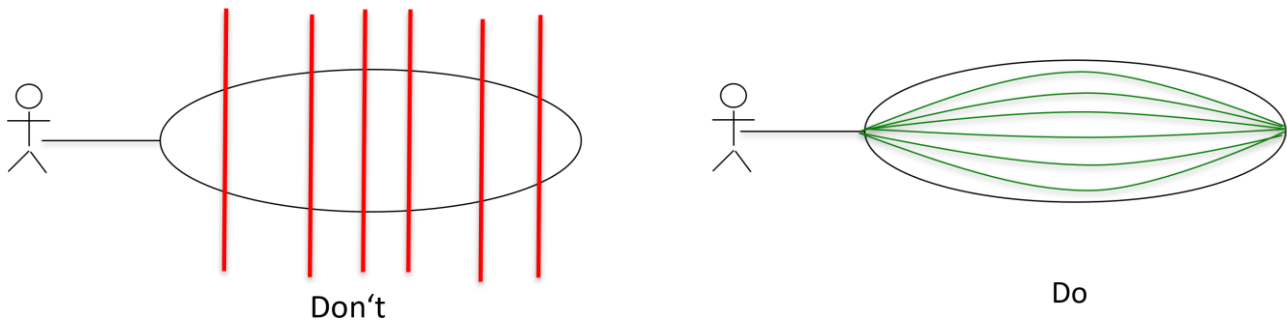


Figure 10: Use case slices instead of process steps

Slicing can be done by different business objects or by technology. Then you can pick one of the slices for early implementation and add others later. In addition you can shrink a slice by:

1. leaving out alternatives (for example first go for the normal flow, adding exceptional cases later on),
2. leaving out options (for example leaving out things that are not absolutely necessary to be implemented in an early release) or
3. leaving out steps that can still be done manually in early releases.

If you originally came up with stories that are too small to create business value (especially if they are not independent and not valuable – thus violating parts of the INVEST principle) you should combine some of them or otherwise reformulate them to get good, even if large, starting stories.

Take a look at the following stories from our case study:

- ▶ As a student I want to enter my name and address so that I can create an account.
- ▶ As a student I want to add my email address to my account to receive a link to the course.

This is too low level for valuable stories since the business rule requires all this data to create an account. Better to reformulate:

- ▶ As a student I want to create an account to get access to the video learning platform.

Decomposition and grouping of stories will result in requirements hierarchies as discussed in chapter 3.1. This hierarchy can be visualized as a two-dimensional story map [Patton2014], see Figure 11. Above the separation line, bigger groupings (like large stories, epics and features) are aligned in a way that tells the complete story of the product. This helps to maintain an overview of the requirements; below the separation line one can attach all lower-level details for the bigger groups and order them for assignment to sprints and releases as in a linear backlog. In other words, the story map shows backlogs per feature or epic while keeping the higher-level structure of the requirements intact.

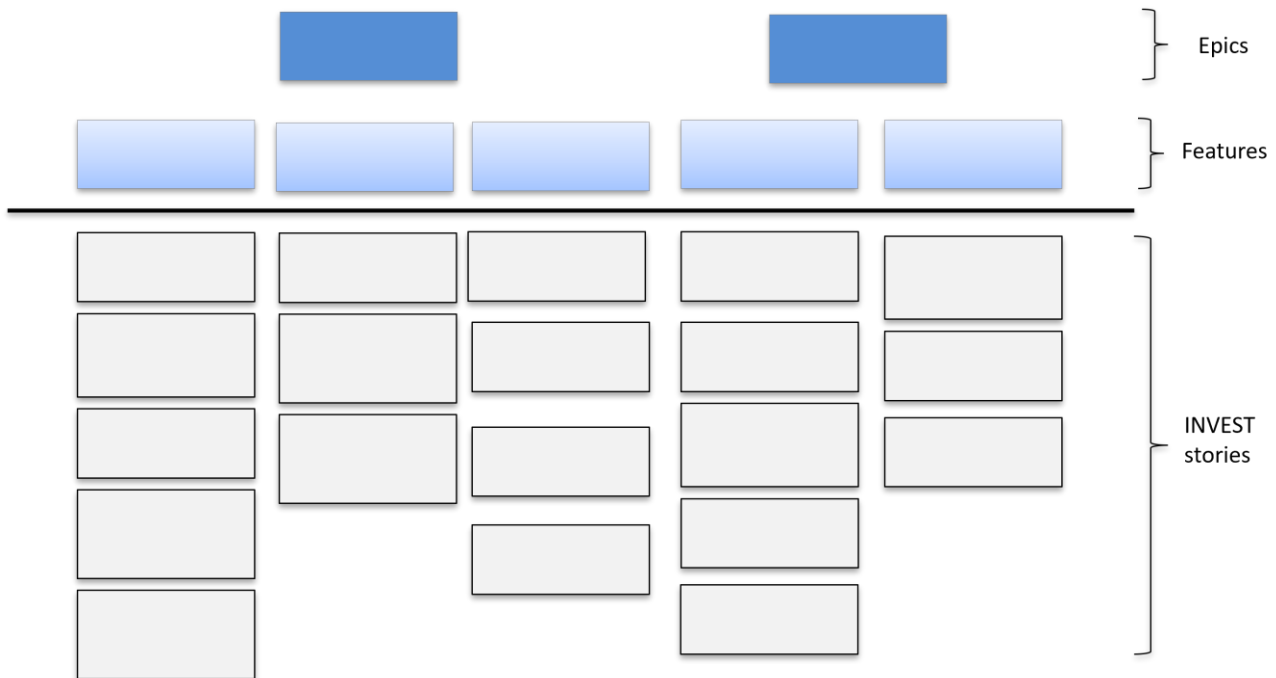


Figure 11: The structure of a story map

3.5 Knowing When to Stop

The Product Owner is responsible for continuing discussions with developers until both sides have a common understanding of the requirements [Meyer2014]. The Pareto principle can be used in assessing when this point has been reached: requirements must not be defined 100% perfectly, but well enough to address the team's key questions and clear enough allowing for the implementation effort to be estimated. Starting the implementation with too many open questions may reduce development speed considerably and cause delays against forecasts.

For this level of joint understanding agile has defined the definition of ready (DoR) [AgileAlliance].

A story is ready when it fulfills the INVEST criteria [Wake2003], especially the last three of the letters:

- ▶ The developers have been able to estimate the story.
- ▶ The estimation is small enough to allow the story to fit into one iteration.
Lawrence suggests that the story should not only fit in one iteration, but it should be so small that 6 - 10 stories can be assigned to the next iteration [Lawrence1]. To achieve this the Product Owner has to be aware of the velocity of the team. (For more details on the velocity see chapter 5.) If for example the team can handle 28 story points per sprint, then the user stories should be so small that the sum of 6 to 10 stories does not exceed that value. The sprint backlog should be composed of for instance 8 stories with 1, 1, 2, 3, 5, 8 and 8 story points and a clear sprint goal should be formulated just in case the team cannot finish all the stories.
- ▶ The Product Owner provided acceptance criteria for the story. Based on the CCC principle everyone agrees that there has been enough *conversation* and that the criteria for *confirmation* of success in terms of acceptance tests were defined. If one uses *cards* to capture the stories the acceptance tests are normally written on the back of the card.

Product Owners have a choice in case of a story that is already small enough to fit into one sprint: they can keep that story and add more acceptance tests to the card. Or they can choose to split the story into multiple stories, usually having less and more primitive acceptance tests for each of them.

Different styles are available [Beck2002] when formulating acceptance criteria. They can be informal natural language sentences to be checked after implementation.

The acceptance criteria could be a little bit more formal using the Gherkin syntax [WyHT2017]. Gherkin is a business readable, domain specific language created especially for the description of behavior. It gives you the ability to remove logical details from behavior tests.

Gherkin suggests the following structure for writing test scenarios:

- **Scenario:** <<short descriptive name>>
- **Given** <<some precondition>>
- **And** <<some other precondition>>
- **When** <<some action by the user>>
- **And** <<some other action>>
- **Then** <<some testable outcome is achieved>>
- **And** <<something else we can check happens too>>

Some methods even advocate using TDD (Test Driven Development). Instead of using a Domain Specific Language (DSL) like Gherkin you can formally code the test cases so that they can automatically be executed after implementation [Meyer2014]. This formal approach – while very precise – may be hard to do and hard to understand for Product Owners and business-oriented stakeholders.

For the Product Owner the DoR is the equivalent to the definition of done (DoD) of the developers. DoD defines criteria to determine whether a story has been successfully implemented while DoR defines that the developers have sufficient information about a user story so that it can be “Done” by the developers within one iteration.

Discussing requirements with developers needs time and is best done prior to the iteration planning. Planning can then focus on selecting the right user stories and assigning these to the responsible developers. Ideally, developers will have seen the requirements evolve, and helped the Product Owner by asking questions and performing estimations.

Different forms of refinements are possible. Refinement meetings may, for example, be a more efficient way of performing refinement than repeatedly disturbing individual developers. The product backlog refinement and all the surrounding activities consume time from the overall iteration capacity. The Scrum guide [S@S Guide] recommends a maximum of 10% capacity from the developers for refinement: if more time than that is required, this is a warning sign for poor quality of the requirements. A Product Owner should understand the relationship between iteration length, risk and iteration overhead, and know that there are shorter feedback loops than the iteration itself.

3.6 Project and Product Documentation of Requirements

Agile projects, especially Scrum ones, use a product backlog, which is a prioritized list of the functionality to be developed in a product or service. Although product backlog items can be whatever the team desires, epics, features and user stories have emerged as the most popular forms of product backlog items.

While a product backlog can be thought of as a replacement for the requirements document of a traditional project, it is important to remember that the written part of an agile user story (“As a user, I want ...”) is incomplete until the discussions about that story has taken place.

It is often best to think of the written part as a pointer to a more precise representation of that requirement. User stories could point to a diagram depicting a workflow, a spreadsheet showing how to perform a calculation, or any other artifact the Product Owner or team desires.

In the RE@Agile Primer [Primer2017] we have identified four different purposes for requirements documentation.

Let us consider the first two purposes:

- a) **Documentation for communication purposes:** Effective and efficient communication is an important tool in agile methods because of its interactivity and short feedback cycles. In practice, there are several situations that may hinder direct verbal communication: distributed teams, language barriers or time restrictions of those involved. Furthermore, information is sometimes so complex that direct communication may be inefficient or misleading. A paper prototype or a diagram of a complicated algorithm can, for example, be re-read later on. Sometimes stakeholders simply prefer written communication to reading source code or reviewing software. In these cases, documentation facilitates the communication process between all involved parties and the results of the process are stored.

The principle of creating documentation for communication purposes is: a document is created as an additional means of communication if stakeholders or the developers see value in the existence of the document. The document should be archived when the communication has been successful.

- b) **Documentation for thinking purposes:** An often-forgotten aspect of writing a document is that writing is always a means to improve and support the thought processes of the writer. Even if the document will be thrown away later in the process, the benefit of improving and supporting thinking is lasting. For example, writing a use case forces the writer to think about concrete interactions between the system and the actors including, for example, exceptions and alternative scenarios. Writing a use case can therefore be understood as a tool to test your own knowledge and understanding of a system.

The principle for creating documentation for thinking purposes is: the thinker decides on the document form that supports his or her thinking best. The thinker does not need to justify this decision. The document may be discarded when the thinking process is finished.

For the first two purposes a product backlog with epics and stories (in whatever form (cards on the wall or stories captured in tools) and maybe augmented with sketches, diagrams and prototypes) is sufficient as documentation to support the progress of product development.

For the two other purposes, more formal requirements documentation must be considered.

- c) **Documentation for legal purposes:** Certain domains or project contexts (for example software in the health care sector or avionics) require documentation of certain information (for example requirements and test cases of a system) to obtain legal approval.

The principle of creating documentation for legal purposes is: the applicable laws and standards describe what legally necessary documentation has to be created. This documentation is an inseparable part of the product.

- d) **Documentation for preservation purposes:** Certain information about a system has a lasting value beyond the initial development effort. Examples include the goals of the system, the central use cases it supports or decisions that were made during its development, for example to exclude certain functionalities. Documentation for preservation purposes can become the shared archive of the team, of a product or of an organization. It can reduce the dependency on the memory capacity of the individual team members and can help discussions about previous decisions (for example "Why did we decide not to implement this?").

The principle of creating documentation for preservation purposes is: the team decides on what to document for preservation purposes.

For these two purposes the product backlog – which is a tool for the interaction of a Product Owner with developers – is not sufficient. The good news is that documentation for legal purposes or for the preservation of product requirements know-how does not have to be created upfront.

It can be updated and maintained every time a new version of the product is released, for instance after the successful implementation of features. Thus, it only contains documentation of functionality and qualities that really made it into the product – avoiding time-consuming version and configuration management activities on documents while stakeholders are still negotiating and maybe changing their opinions.

Defining an adequate degree of documentation depends on many factors like the size of the projects, the number of stakeholders involved, legal constraints, and/or safety-critical aspects of the product. Based on these factors, agile teams try to avoid documentation overkill and find a minimum set of useful documentation.

While working with a “living” product backlog is an efficient way to handle documentation, it is not always sufficient. A structured up-to-date documentation of all requirements implemented in a product may not only be a legal constraint in some projects but also a perfect starting point for quicker identification of change requests based on existing documentation.

3.7 Summary

Whatever your stakeholders tell you about required functionality is the right starting point for requirements work. But it is the starting point only. Your job as Product Owner is to bring structure into these functional requirements.

Epics, themes, features or large stories (representing potentially complex business processes) are a good way to keep a big picture, an overview of all the things that your stakeholders want from a system or a product. But you have learned that – by definition – they may not be precise enough to stop at that level.

Your goal for good requirements work is to come up with user stories, that fulfill the definition of ready, or the INVEST criteria: they should be independent and valuable, small enough to fit into one iteration, estimable and equipped with testable fit criteria. Mike Cohn’s template “As a <user> I want <some functionality> to achieve <some goals>” is a good starting point, but you should not insist on using this formula in all cases.

If a requirement is still too large to fit into one iteration you have learned several tactics to split them, while you still try to preserve independence and value as much as possible.

4. Handling Quality Requirements and Constraints

Chapter 3 focused on handling Functional Requirements. Dealing with Functional Requirements, meaning finding out what functionality the various stakeholders need, will be the most time-consuming activity in system development and it will dominate most discussions between Product Owner, stakeholders and the developers.

Qualities of (the functions of) the system, like performance, user friendliness, robustness and extensibility are often taken for granted. Users and/or other stakeholders often assume that they do not have to be stated explicitly since the developers already know about them.

The same is true of organizational and technical constraints. Doesn't everybody know that we have a standard process model, requiring certain artifacts to be produced? Isn't everybody aware that we always use company X to buy our database systems, and of course will code in language Y?

Requirements Engineering experts have asserted the importance of these "non-functional" requirements for decades. Even though the term "non-functional requirements" is still often used in practice, as an umbrella term for quality requirements and constraints, IREB uses the more concrete and precise categories "Quality Requirements" and "Constraints", according to [Glinz2014].

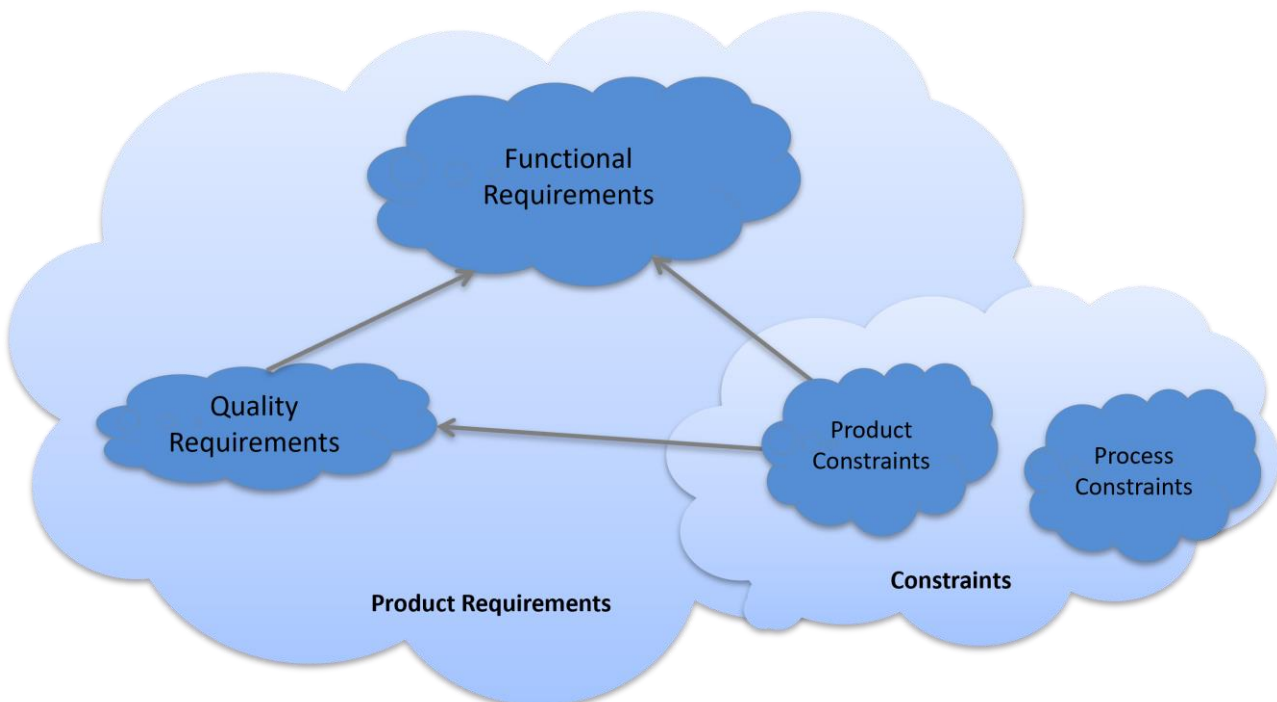


Figure 12: Categorization of requirements

Figure 12 shows the three categories of requirements and some of their important relationships. A quality requirement will never stand-alone, meaning that it will always refer to one or more - or even all - functional requirements. Constraints are either product constraints, constraining the design of a function or a quality, or process constraints, restricting the work of the developers in a way that is not directly linked to the product itself, for instance certain process steps have to be performed or certain artifacts have to be created.

Initially quality requirements and constraints are often deliberately vague. In the next chapters we will describe how to capture such vague qualities and constraints. You will also see how to transform vague quality requirements and constraints into more precise requirements (down to the level of specifying precise acceptance criteria) and how to handle them in conjunction with functional requirements.

4.1 Understanding the Importance of Quality Requirements and Constraints

[Meyer2014] expresses the concern that “many agile methods concentrate on functional requirements only and do not put enough emphasis on qualities and constraints”. Bertrand Meyer goes on to say: “Key constraints and some categories of qualities envisaged for the system should be made explicit early in the lifecycle of a product, since they determine key architectural choices (infrastructure, software architecture and software design). Ignoring them or learning too late in the project may endanger the whole development effort. Other qualities can be captured iteratively, just in time, as with functional requirements.”

While there are many categories of quality requirements to be considered, the task is made somewhat easier for Product Owners by a number of published categorization schemata – or checklists – such as those shown in the two following examples. As a Product Owner you should simply use one of these “cheat sheets” to ask explicit questions about these qualities. Even better: based on the available checklists you can create your own checklist to emphasize the qualities that are most important in your domain.

In 2011 ISO published a new quality standards family, replacing the well-known ISO/IEC 9126 quality model from 2001. The most important standard for Requirements Engineering is [ISO25010], defining quality requirements. Its latest update is from 2017. Figure 13 shows the eight top-level quality characteristics of systems and their decomposition into sub characteristics. Note that the standard does not talk about requirements, but about system qualities. Adding the word “requirements” to each category allows you to discuss your needs in this area, for instance “capacity” becomes “capacity requirements”.

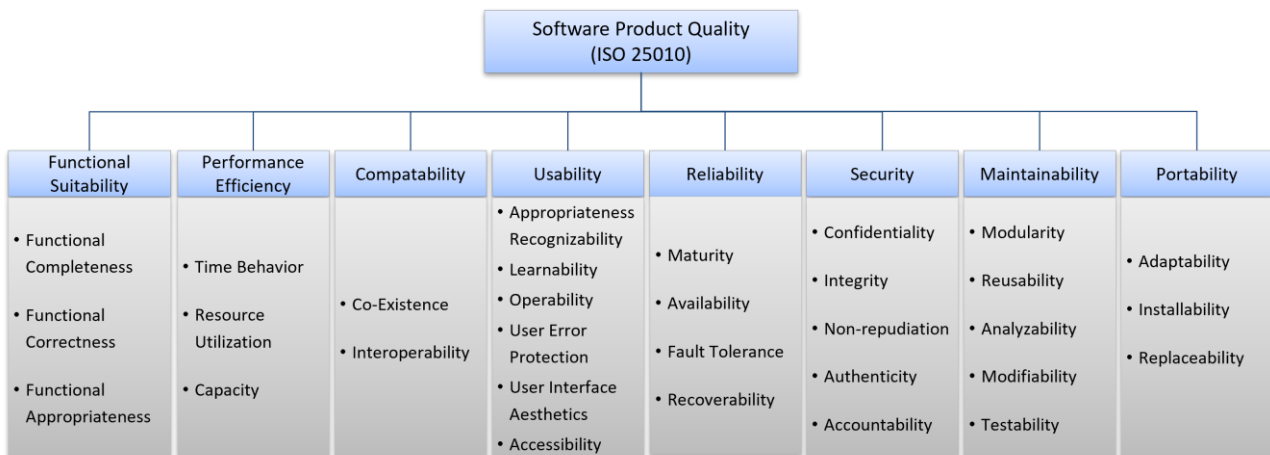


Figure 13: Categories of qualities according to ISO25010

Detailed definitions of all these categories can be found in the standard. In addition to the generic quality model the ISO/IEC 25012 standard contains a complementary model for data quality.

A similar categorization scheme for quality requirements can be found in the VOLERE template [RoRo2017]. Chapters 10 – 17 of this template describe categories of quality requirements. The categorization is based on decades of experience in system specification. The original template adds the word “requirements” to every category, i.e. “longevity” reads “longevity requirements”. In Figure 14 we have skipped this addition to keep the categories more readable.

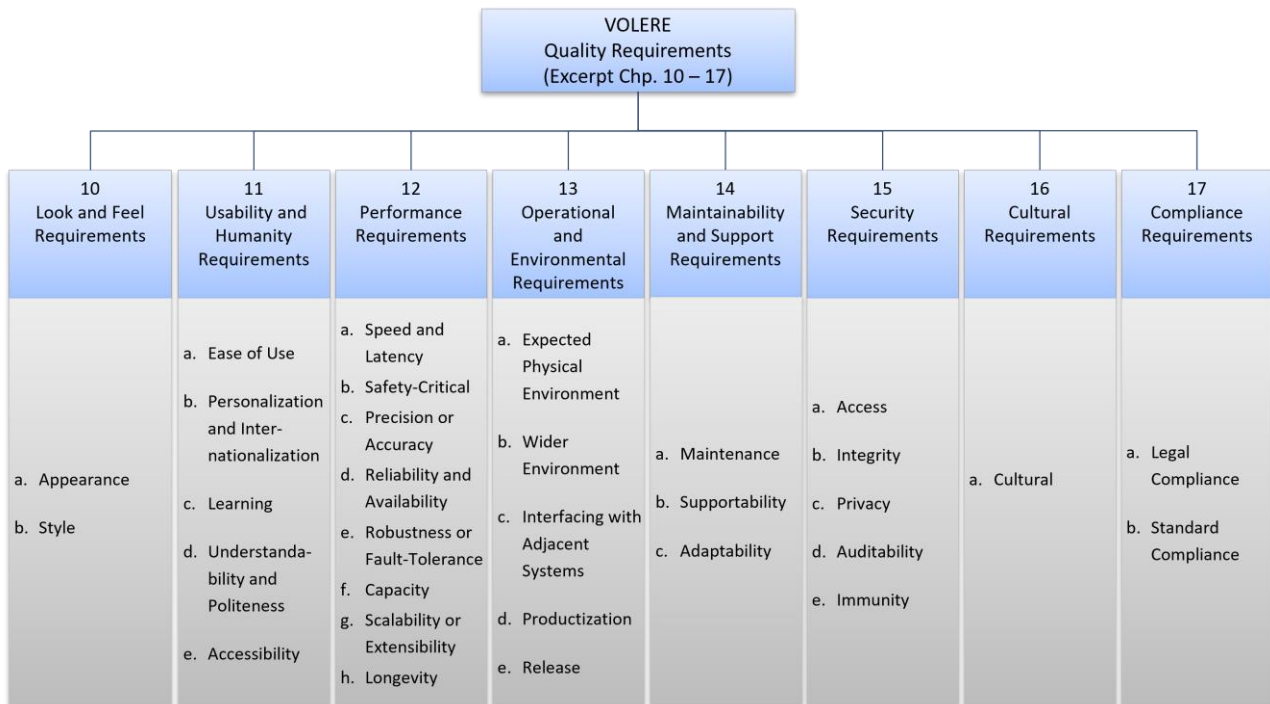


Figure 14: Quality categories of VOLERE

In [RoRo2013] you will not only find definitions of all these categories, but also the reason why they are important. You will also find examples of how to formulate them including acceptance criteria.

The following example is taken from <http://volere.co.uk/template.htm> [RoRo2017]. Note that acceptance criteria are called fit criteria in this publication.

11c. Learning Requirements

Content

Requirements specifying how easy it should be to learn to use the product. This learning curve ranges from zero time for products intended for placement in the public domain (for example a parking meter or a web site) to a considerable amount of time for complex, highly technical products.

Motivation

To quantify the amount of time that your client feels is acceptable before a user can successfully use the product. This requirement guides designers in understanding how users will learn the product. For example, designers may build elaborate interactive help facilities into the product or the product may be packaged with a tutorial. Alternatively, the product may have to be constructed so that all of its functionality is apparent upon first encountering it.

Examples

The product shall be easy for an engineer to learn.

A clerk shall be able to be productive within a short time.

The product shall be able to be used by members of the public who will receive no training before using it.

The product shall be used by engineers who will attend five weeks of training before using the product.

Fit Criterion

An engineer shall produce a [specified result] within [specified time] when beginning to use the product, without having to use the manual.

After receiving [number of hours] training a clerk shall be able to produce [quantity of specified outputs] per [unit of time].

[Agreed percentage] of a test panel shall successfully complete [specified task] within [specified time limit].

The engineers shall achieve [agreed percentage] pass rate of the final examination of the training.

Suggestions for Exercise:

Discuss for some of the categories shown in Figure 13 or Figure 14 whether the developers should know about these requirements early on or if they can be considered later in the development process.

4.2 Adding Precision to Quality Requirements

Quality requirements have to be communicated to the developers in a way that is both unambiguous and testable. As mentioned earlier, quality requirements are often very vague at the beginning. For example: *The new mobile phone generation shall be attractive to teenage kids.*

This quality requirement is neither unambiguous nor testable (in the way it is expressed), but might nevertheless be the starting point for discussions about more detailed qualities required for the next generation of mobile phones.

Its precision (or rather lack of) can be compared to a functional epic like “As a mobile phone user I want intelligent dialing capabilities”. In chapter 3 we discussed how to bring such an epic to the level of precision allowing for the developers to implement it.

In this chapter we will do the same for quality requirements. We will first explain how to make quality requirements more concrete, down to the level of having acceptance criteria. Then –in chapter 4.3- we will describe how and where to (physically) record or store them.

There are two ways of adding precision and clarity to vague quality requirements. You can either detail or decompose them, or you can derive more precise (functional) requirements from the original requirement. Figure 15 graphically shows these alternatives.

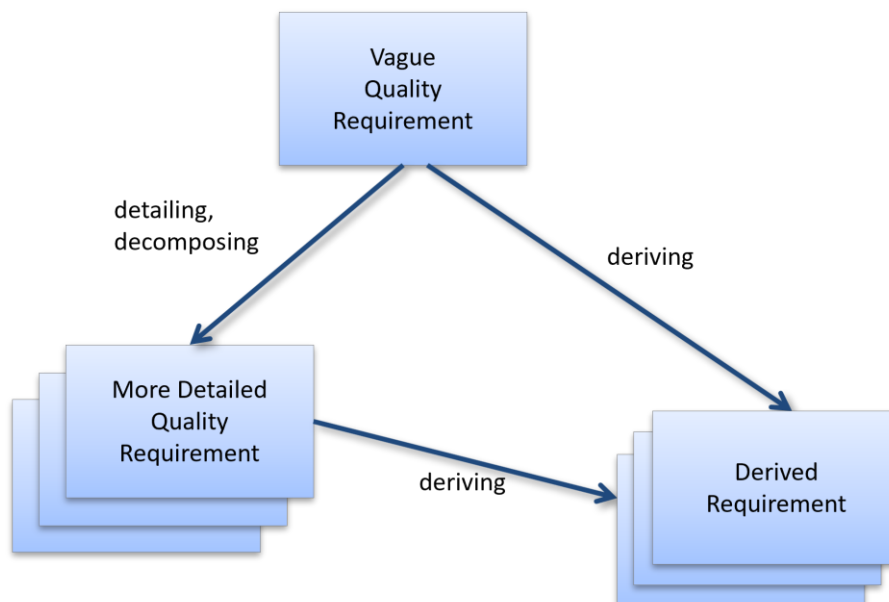


Figure 15: Detailing and decomposing quality requirements

Detailing or decomposing takes the original vague quality requirement and replaces it with two or more detailed quality requirements.

Example: Looking at the categorization schema in Figure 14, you could detail the usability requirement (VOLERE category 11) *“The system should be user friendly”* with the following two requirements:

- ▶ As a user I want the system to be easy to learn (VOLERE category 11c), and
- ▶ As a user I want the system to be easy to handle (VOLERE category 11a).

These two are still vague but already more precise than the original one.

The second alternative “deriving” means to transform the original quality requirement into one or more (functional) requirements.

Take for example the original requirement: *“As security officer I want the access to the following functions restricted to authorized personnel.”*

Deriving more precise requirements means for example deciding that a login mechanism with user name and password will be used to restrict the access.

Note that the original intention of the quality requirement was just to secure the access to certain functions. It is a design decision to achieve this by introducing roles and passwords. You could come up with other ideas, like locking away the computer in a room to which only authorized persons have access. Alternatively, you could decide to use fingerprints to identify authorized users.

If you derive new functional requirements from original quality requirements you might want to keep the original requirement, for instance to remember its origin, in case in future versions of the product you discover more clever ways to achieve the original quality. Deriving new functional requirements from required qualities brings you closer to a solution or a fulfillment of that requirement.

Suggestions for Exercise:

Pick one of your products and refine some examples of quality requirements.

Quality trees [Clements et al.2001] are also a proven way to structure quality requirements. A quality tree combines the two techniques mentioned above. Figure 16 shows the generic form of a quality tree. It starts with a root labeled *“specific quality”*. The next branches of the tree are categories of qualities, followed by subcategories. The leaves of the tree show concrete scenarios for a category or subcategory, for instance functional requirements or testable quality statements.

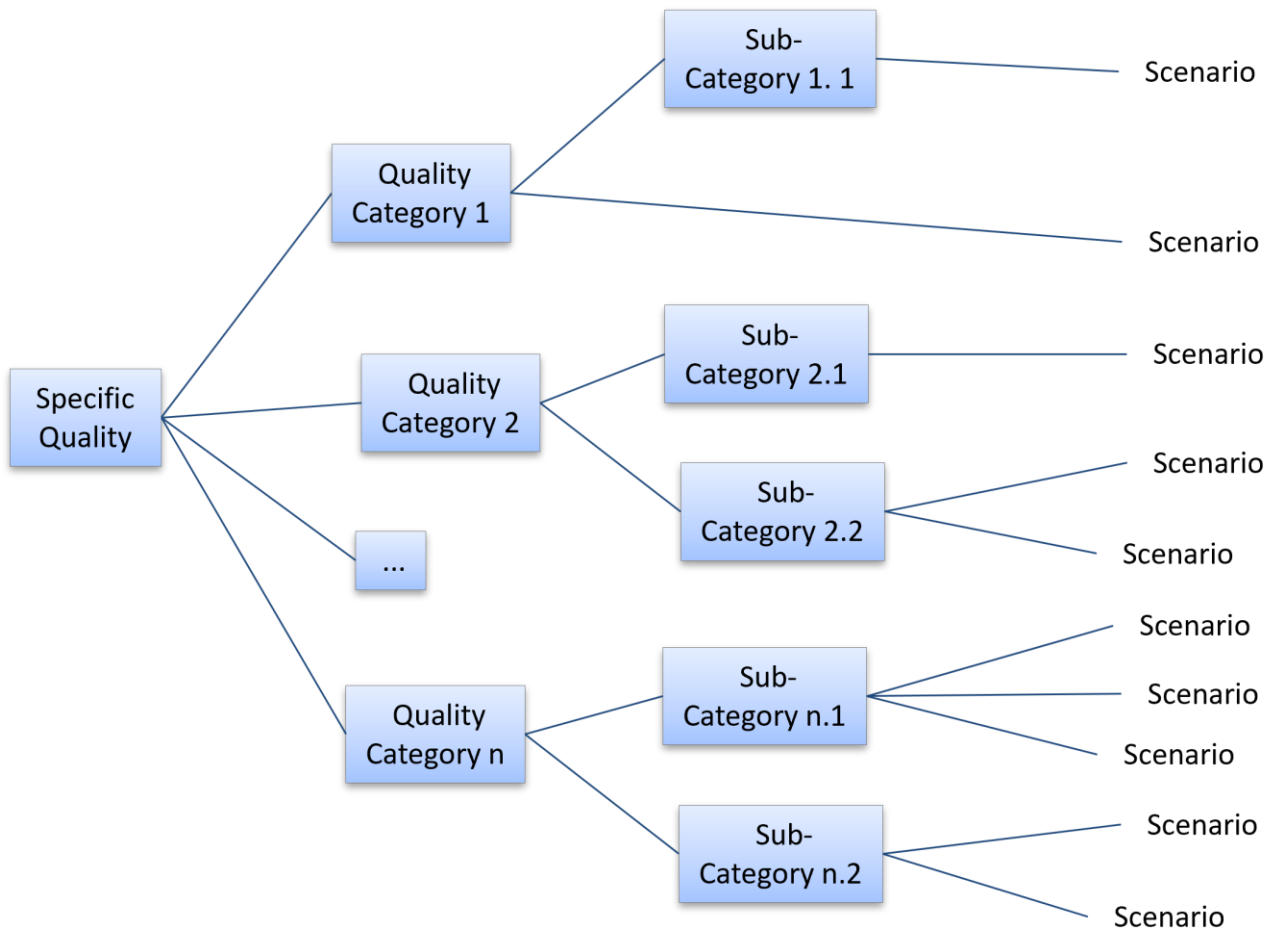


Figure 16: A generic schema for a quality tree

For our case study iLearn Figure 17 shows excerpts from a quality tree. Note the following points:

- ▶ The leaves may still not be precise enough to be tested, for example: “usable without training of students”. That is why quality requirements need acceptance criteria to inform the developers about the expectations of the Product Owner.
- ▶ There is a very clear business decision in the requirement for “other languages”. The Product Owner, together with all stakeholders, has decided that subtitles are sufficient for marketing the product in other countries, rather than, for example, dubbing the videos.
- ▶ There is even a design suggestion in the “adaptability” requirement: instead of just asking that the system should work on various kinds or devices with different resolutions, the Product Owner requests use of the corporate standard technology: responsive design.

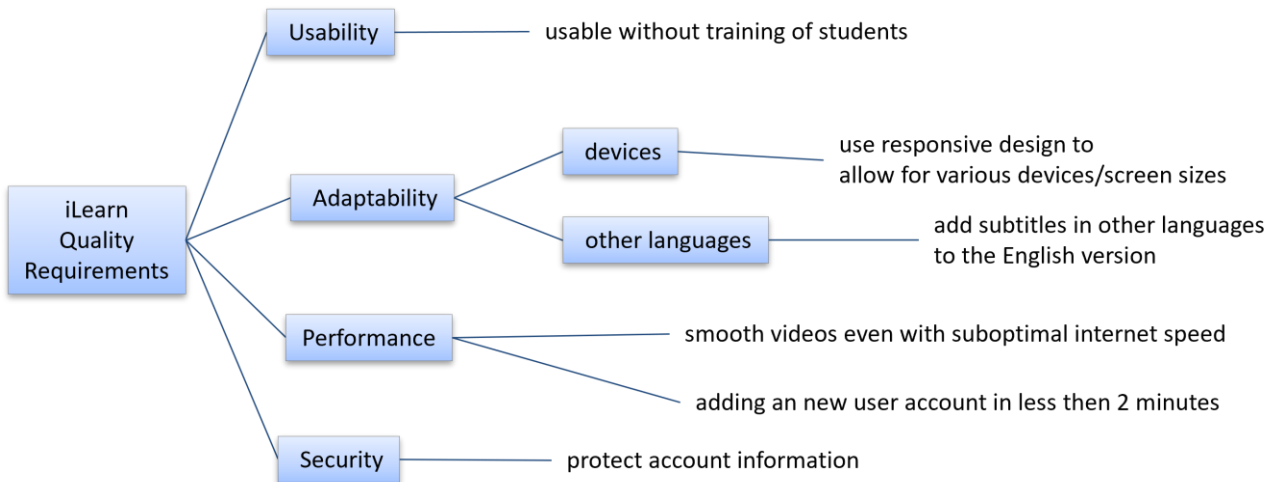


Figure 17: Parts of a quality tree for iLearn

Suggestions for Exercise:

Try to brainstorm on a partial quality tree for one of your products. Make sure that you have very concrete scenarios as leaves!

As mentioned earlier, quality requirements also need acceptance criteria to add more precision. The type of acceptance criteria used will depend on the category of the quality. The following table shows systematic advice on how to formulate acceptance criteria for different VOLERE categories of qualities.

Req. Type	Suggested Scale
10 Look & Feel	Conformance to standard - specify who/how this is tested
11 Usability	Amount of learning time Amount of training Test panel can perform functions in target time
12 Performance	Time to complete action
13 Operational	Quantification of time/ease of use in environment
14 Maintainability	Quantification of portability effort Specification of time allowed to make changes
15 Security	Specification of who can use the product, and when
16 Cultural & Political	Who accepts, quantification of special customs
17 Legal	Lawyer's opinion / court case

The following chapters provide examples of acceptance criteria for quality requirements. More information can be found in [RoRo2013].

Usability Requirement: The product must be useable by a member of the public, who may not speak English.

Acceptance Criterion: 45 out of 50 randomly selected non-English speakers must be able to use the product within the performance criteria plus 25%.

Performance Requirement: The product must be acceptably fast.

Acceptance Criterion: Each transaction at the vending machine must take no more than 15 seconds.

Operational Requirement: As a worker I have to use the product also when outside in cold, rainy conditions.

Acceptance Criterion: 90% of workers in the first month of use must successfully use the product within the target time constraints.

Security Requirements: Only direct managers may see the personnel records of their staff. Personnel records of staff may not be viewed by anyone else.

Acceptance Criterion: Recording the accesses and testing to see if a non-manager had access. Alternatively, you might say that the product must be certified as conforming to the xyz-security standard.

Legal Requirement: Personal customer information must be used in accordance with the Data Protection Act.

Acceptance Criterion: The legal department must agree that the product conforms to the organization's data protection registration.

Suggestions for Exercise:

Pick two examples of quality requirements and add acceptance criteria to them.

4.3 Quality Requirements and Backlog

We discussed how to discover and elicit quality requirements and how to make vague quality requirements more precise. Now we will discuss how to document them in an agile environment in conjunction with a product backlog containing mainly functional requirements. Depending on the kind of quality requirement, one or other of the following approaches will work for you.

The easiest way to record a quality requirement is to attach it directly to a backlog item. This approach only works if the quality is unique to that one feature or user story.

A second approach is to record quality requirements outside the backlog, either:

- ▶ On separate cards;
- ▶ As a quality tree.

In both cases you have to link them to all the relevant functional requirements. Depending on the tools you use this may be done either using hyperlinks, or you have to explicitly enumerate the functional requirements targeted by each quality.

The third alternative is to put quality requirements in the definition of done. Since the rules in the definition of done apply to ALL iterations, you are indicating that you always want that requirement to be obeyed, independent of which functional requirements you attach to the next iteration.

4.4 Making Constraints Explicit

Constraints are an important type of requirements. Glinz defines constraints as requirements that limit the solution space beyond what is necessary for meeting the given functional requirements and quality requirements [Glinz2014]. The product must be built within the constraints. Constraints restrict what you are allowed to decide and thus influence and shape the product.

They are either determined by your management or by other stakeholders outside your scope of control, for example regulatory authorities, your parent company or an enterprise architect.

Note that while many constraints are certainly legitimate, it is often worthwhile for the Product Owner or developers to check their validity and to negotiate with persons or organizations that put such constraints on your development; to question their reasons and motivations.

Sometimes you will discover that some of the constraints are pure folklore that – once you question them and suggest alternatives - can be negotiated with the responsible stakeholders and relaxed, allowing more flexibility in the implementation. So, in agile terminology: Constraints may also be negotiable, in the same way as functionality. However, if the other parties insist on these constraints, then the developers have to accept them.

In this Handbook we have included legal requirements or (more general) any kind of compliance requirements as categories of quality requirements (see chapter 4.1). They could as well be included in this chapter on constraints since any solution has to have these qualities. Compared to the other categories of constraints such compliance requirements are often non-negotiable.

Figure 12 shows one way to categorize constraints: They can be classified either as product constraints or as process constraints. Only product constraints refer to functional or quality requirements of the product, thus limiting their implementation. Process constraints have no direct relationship to the product. They put limits on the organization that develops the product, or the development process used for the development of the product. Thus, they have only an indirect effect on the product itself.

Figure 18 suggests some sub-categories for these two categories. Some examples are discussed in the following text. More details about how to formulate such constraints, and more examples, can be found in [RoRo2013].

The product constraints may ask for a given infrastructure, meaning a technological and/or physical environment in which the product is to be installed. Other examples include the mandatory use of off-the-shelf software (meaning a buying decision as opposed to developing sub-systems within the project).

Constraints	
Product Constraints (often technological constraints)	Process Constraints (often organizational constraints)
<ul style="list-style-type: none"> • Constraints about the technological environment • Off-the-Shelf Software (Make or Buy) • Reuse of Components • Anticipated Workplace Environment • Prescribed Technology • Physical Constraints • Environmental Constraints 	<ul style="list-style-type: none"> • Schedule Constraints • Budget Constraints • Skill Constraints • Prescribed Process Models (Roles, Activities, Artifacts) • Compliance Regulations • Constraints about Deployment and Migration • Constraints about Support

Figure 18: Categorization of constraints

The constraint to reuse existing components or sub-systems of predecessor products or other products the company developed is one that is often introduced. The reason for reuse is obvious: you don't want to spend money if you have acceptable (partial) solutions at your disposal.

Constraints concerning the anticipated operational environment of the product describe any features of the workplace that could have an effect on the design. Product designers should know, for example, that the workplace is noisy, so audio signals might not work.

Conversely, where the product is intended to operate in quiet environments, the noise level produced by the product should not exceed a certain level of decibels. If the workplace is in the open air where it could be wet and cold, then users should be able to use the product wearing gloves.

Similar for systems involving hardware elements, physical constraints such as those related to the size or weight of the device – think mobile phones or other handheld devices – may also be very relevant (meaning relevant to both the hardware design and to the software which it is able to support).

The most common product constraints, however, limit the technology that the developers are allowed to use.

For example:

- ▶ As enterprise architect, I want you to develop the product in C# so that our existing staff can maintain the product.
- ▶ As database administrator, I want the product team to use ORACLE since we have excellent hotline support for this product.

Note that you don't have to write constraints as stories. It may be sufficient to inform the team that C# and ORACLE are non-negotiable constraints.

Process constraints are often called organizational constraints, since they constrain either management aspects like budget, schedule or the skills of team members available for the project (“You have to work with this team. We have no budget to hire additional staff and no budget for external people.”) or they enforce certain policies and regulations. You might have to follow a development process that prescribes certain roles, mandatory activities to be performed during development and a set of documents or other artifacts to be produced and maintained.

Constraints, like other types of requirements, have a description: they can contain a rationale or motivation describing why the constraint is in place. And they should also have acceptance criteria – just as for functional or quality requirements.

If you have worked in an organization for some time, you are likely to have learned about the technological preferences in the organization and you will be aware of organizational rules and constraints. Nevertheless, it is important to make such constraints explicit so that everyone else in the team is aware of them. The most limiting ones should be known early in the project. Others should be captured as soon as they are discovered.

Such constraints are normally applicable to a wider range of projects. Basic technology stacks, as well as process models, are normally set for a longer period in a company. So as soon as these constraints are captured, they can easily be reused in different product developments.

4.5 Summary

Quality requirements and constraints are as important for project success as functional requirements. For a Product Owner it is not difficult to find relevant requirements in these categories since there are many checklists available in the public domain, suggesting categories for qualities and constraints.

Quality requirements may start out vague. Before being ready for development they have to be made more precise, down to the level of acceptance tests – just as for functional requirements.

Adding precision to quality requirements is often achieved by deriving corresponding functional requirements that fulfill the originally required qualities. Make sure such decisions are recorded, and that the original quality requirements are not discarded, since over time you might discover better ways to fulfill the qualities.

Some quality requirements can simply be attached to already discovered user stories, for example adding performance or special security aspects to individual functions. Many quality requirements concern crosscutting aspects, meaning they are relevant to many of the functional requirements.

For those we suggest that you maintain a separate list, always visible to the developers, since they must always be fulfilled. An alternative is to include them in the definition of done, which has the same effect of being always valid.

A similar approach can be taken for technical, organizational and legal constraints. Make sure they are explicitly known to the developers. If they are not project specific, but more general company rules, you can maintain them in a central location for all projects thus reusing them over many development projects.

5. Prioritizing and Estimating Requirements

Agile approaches aim to maximize the overall business value over time and to permanently optimize the overall business value creation process [Leffingwell2010]. This constant value adding process is shown in Figure 19. Every iteration should result in added value – sometimes more, sometimes less.

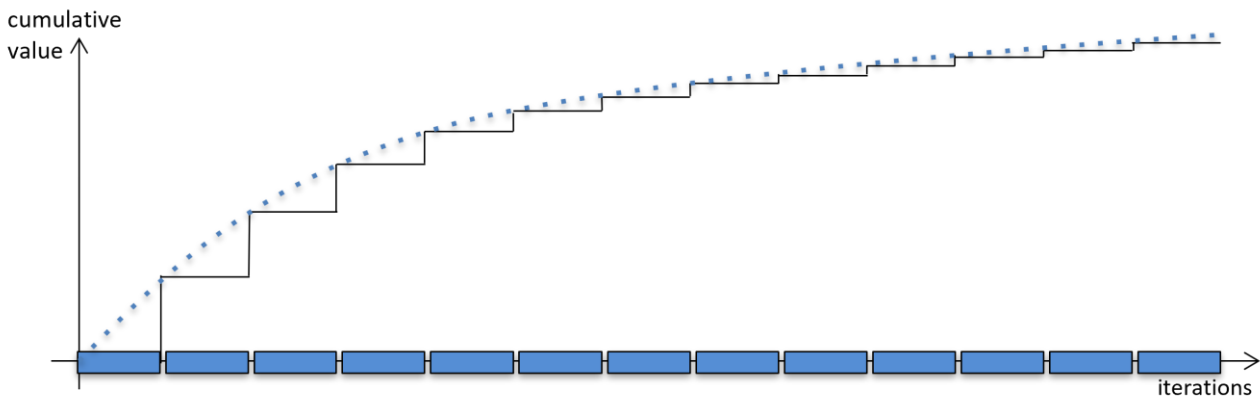


Figure 19: Agile development = constant value creation

Every iteration is supposed to deliver a potentially releasable product increment that increases the value of the overall product. (Comment: some versions of Scrum and other agile approaches refer to a “potentially shippable product” or “potentially usable product increment”).

[LeSS] explains this goal as follows: “Potentially shippable is a statement about the quality of the software and not about the value or the marketability of the software. When a product is potentially shippable then it means that all the work that needs to be done for the currently implemented features has been done and technically the product can be shipped, but it doesn’t mean that the features implemented are valuable enough for the customer to want a new release. The latter is determined by the Product Owner.”

When planning for and achieving this constant addition of value, all requirements (whether coarse or fine) should be ordered primarily based on the added value they can bring to the business. But business value can mean many different things to different organizations. Clarifying this term “business value” is one of the core topics of this chapter and will be discussed in chapters 5.1 to 5.3.

Of course, creating value has to be balanced with the effort to create it and the moment in time when the value will be delivered. Therefore, the developers have to support the Product Owner with estimates about the efforts needed to create the business value. Estimating backlog items is the second core topic of this chapter and will be discussed in chapter 5.4. Based on the value/effort ratio the Product Owner can select the stories that should be taken on by the developers in the next iteration.

5.1 Determination of Business Value

As mentioned above “value” can mean many different things in different environments. Here are some aspects to be considered when establishing business value and when putting the backlog items in order by that value.

- Value to the customer or other stakeholders

If you develop a product for a specific customer or client, the opinion of this client about what is more important and what is less important will definitely influence when you pick backlog items. Not every stakeholder will consider money as a criterion for value. Value for Greenpeace for instance could be anything good you do to protect the environment. So, whatever your customer or important stakeholder values most will be considered.

- ▶ Value to the organization

Despite having specific clients that will use or buy the product the organization itself might (or should) have strategic goals it wants to achieve, for instance create a reusable platform for a given domain, so that future individual projects can be delivered quicker and cheaper. In fact, any kind of optimization and automation of internal business processes can be a driving force for creating value for the organization. If the backlog items are strongly related to such strategic goals, then their business value will be considered as very high.

- ▶ Threat to existence

Not having or offering a certain feature or functionality can be a threat to the product or the overall organization. Typical examples of such threats are legal requirements (for instance data protection). Such a feature may not add business value in a commercial sense, but it must be implemented to ensure to the further existence of the product or the company.

- ▶ Expected financial value of a feature (sales volume, total revenue, return on investment)

Most commercial organizations' goal is to make money (profit). So, features and stories will naturally be ranked higher if they promise more sales or a quick return on investment.

- ▶ Short-term project goals or release goals (versus mid-term product goals)

Sometimes it is important to be able to demonstrate features or at least mockups of features at an upcoming trade show or an important presentation. Therefore, Product Owners may value such results more than those that contribute to the longer-term product strategy. On the other hand, an organization may want to invest in a development framework that does not immediately create business value but reduces long-term development costs and improves the value-add ratio for upcoming product increments.

- ▶ Costs of delay

This is a very interesting criterion to use for determination of business value. The key question is: What is the cost of a delayed shipping of a story? For example, a new feature of an online shopping portal is supposed to increase sales volume by \$500,000 per month means that the company loses \$500,000 if the feature is delayed for one month. Reinertsen [Reinertsen2008] considers cost of delay as a point of view that can summarize all the other aspects mentioned in this chapter.

- ▶ Time to market

Certain features may come with a window of opportunity. For example: If this feature is available within this period, then it will create a significant increase in business. If it comes too late the value might be significantly lower. For example, trade shows are a good opportunity to sell new products to the market. If the product is not ready when the trade show opens, then the customers may buy another product and will have no need to buy the product in the near future even if the product has more and better functionality. Some methods therefore suggest putting an attribute on each backlog item specifying "best before". This way every stakeholder explicitly knows about the window of opportunity.

- ▶ Requirements frequency

If you develop a product for a mass market it may be important to get an understanding of the demand when determining the business value. Did many customers ask for it? Or was it just a small group? How much revenue do you expect to make based on the number of customers that requested the feature?

- ▶ Business dependencies and technical dependencies

Sometimes you have to prioritize a backlog element because it is a prerequisite for one or more other backlog items, meaning the other items cannot be developed if this one is not available. An example in the iLearn case study would be: the development of a user account does not create business value, but you cannot develop personalized features if you have not yet developed the user account feature. These dependencies could also be technical dependencies, for instance developing a feature requires the establishment of a certain infrastructure or certain tools have to be bought and explored before you can deliver the feature. These prerequisites (features) will not create business value, but without having these prerequisites done, you cannot develop the really valuable backlog items.

Also, some of the qualities might be considered to have high value. You might prioritize backlog items that for instance:

- ▶ Improve usability;
- ▶ Improve robustness;
- ▶ Reduce maintenance costs;
- ▶ Minimize impact on the current system.

Working on such quality improvements does not often create new sellable features, so they don't create direct revenue. But they may be considered to be very important by certain groups of stakeholders and therefore be high in the ranking of backlog items.

The delivered value can only be measured on the side of the end user because the end user of the product will decide if they want to use (and buy) the product and if they will recommend the product to other potentially customers. As a result of this the revenue of the producing company may increase.

If the customer is internal there is no revenue to measure so typically the value of the delivered product increments is determined by rating the delivered product increment and the resulting product version sprint by sprint and comparing it to the product roadmap based on the planned and delivered features and product capabilities.

5.2 Business Value, Risk

An important criterion to prioritize backlog items is that some are riskier than others.

[DeMaLi2003] gives a cyclic definition of risks and problem:

- ▶ A risk is a potential problem.
- ▶ A problem is a risk that has manifested itself.

There are many categories of risks in product development. The feature itself could be risky, because for example it may not be accepted by the target audience. The risk could be in the implementation of a feature, for instance if the team wants to use certain technology whereas not all team members are proficient with the technology.

Or the risk could be in the technology itself, which may be too new (and therefore dangerous to use) or too old or outdated. For a comprehensive overview of risks, especially the five main risks that impact every IT project, we refer to [DeMaLi2003].

Maybe the risky backlog items don't deliver high business value based on the criteria defined in the last chapter. But if you want to handle the risks in order to avoid surprises later on, then you may want to pick backlog items that come with a risk early on in the development process. Once you dealt with those items the rest of the work is less risky.

There are four alternatives you can choose from when you have risky backlog items:

1. **Avoid the risk:** This means not handling backlog items that are risky. Avoiding such items implies missing out on the opportunities associated with the items. So avoiding should not be your choice in dealing with risky items.
2. **Mitigate risks:** As a manager you can put money and/or time aside to handle risks as soon as they become problems. As a Product Owner (responsible for Requirements Engineering) you may therefore postpone the detailed study of such items until they become important for the business.
3. **Reduce risks:** besides mitigation this is your second obvious choice to deal with risky items. But this means to take actions now in order to reduce the risk. You typically break down a risky item into smaller items (for example spikes) that allow you to learn more about their risky parts. For instance, you develop a UI-prototype to ensure that the target audience will accept it, or you develop a prototype to gain experience with a new framework.
4. **Hope that the risk does not turn into a problem.** Similar to the first alternative this is not a feasible choice. Imagine that you have twelve risks with a probability of only ten percent each. Mathematics shows that the chance that one of these will hit you is already 75 percent.

As a Product Owner you only want to go for alternatives two and three. From a requirements point of view alternative three is the most important one. You have to find ways to decompose a requirement in a way that reduces the risk. Sometimes you might study a spike or develop a prototype to reduce the risk before moving towards actual feature development.

[DeMaLi2003] concludes: "The real reason we need to do risk management is not to avoid risks, but to enable aggressive risk-taking."

Suggestions for Exercise:

Discuss what (combination of) criteria are used in your organization to determine (business) value.

5.3 Expressing Priorities and Ordering the Backlog

Once you have determined what value means to you, you have to express these priorities and order the backlog according to the priorities given to the backlog items. There are many different methods to assign value to backlog items, some of them very simple, others highly complex. In the following chapter we will discuss popular approaches.

One method is to use MoSCoW. This prioritization method was developed by [ClBa1994] to reach a common understanding with stakeholders on the importance they place on the delivery of each requirement. The term MoSCoW itself is an acronym derived from the first letter of each of four prioritization categories (Must have, Should have, Could have, and Won't have), with the interstitial o's added to make the word pronounceable.

The categories are typically understood as:

- ▶ **Must have:** Requirements labeled as *Must have* are critical to the current delivery time box in order for it to be a success. If even one *Must have* requirement is not included, then the project delivery should be considered a failure (note: requirements can be downgraded from *Must have*, by agreement with all relevant stakeholders; for example, when new requirements are deemed more important).
- ▶ **Should have:** Requirements labeled as *Should have* are important but not necessary for delivery in the current delivery time box. While *Should have* requirements can be as important as *Must have*, they are often not as time-critical or there may be another way to satisfy the requirement, so that it can be held back until a future delivery time box.
- ▶ **Could have:** Requirements labeled as *Could have* are desirable but not necessary and could improve user experience or customer satisfaction for little development cost. These will typically be included if time and resources permit.
- ▶ **Won't have (this time):** Requirements labeled as *Won't have* have been agreed by stakeholders as the least-critical, lowest-payback items, or not appropriate at that time. As a result, *Won't have* requirements are not planned into the schedule for the next delivery time box. *Won't have* requirements are either dropped or reconsidered for inclusion in a later time box.

A simpler schema for expressing priorities could be to use three categories (instead of the four of MoSCoW), labeled H(igh), M(edium) and L(ow) or alternatively A, B and C.

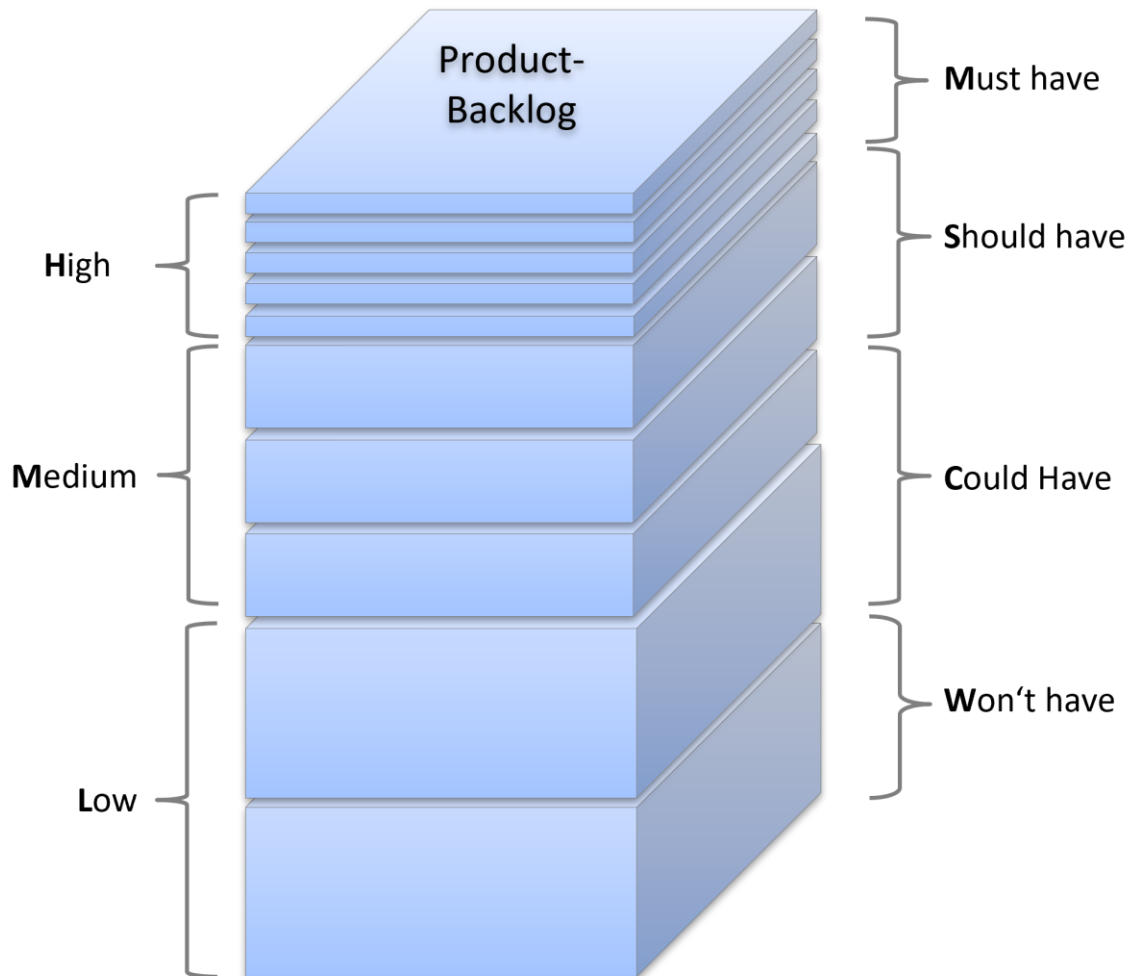


Figure 20: MoSCoW or high/medium/low priorities

Figure 20 shows a backlog where the items are annotated with high, medium and low or MoSCoW. Note that the higher the value given to the requirement the more detailed it should already be described, since it is a potential candidate for the next (or one of the next) iteration(s).

Some companies use a range of numbers between 1 and 100, interpreting it in a way that a higher number means more business value. Thus, you can express bigger differences for instance by giving priority 87 to one backlog item and 38 to another, clearly indicating how much more important the item with priority 87 is.

Figure 21 shows a range of numbers given to smaller or larger backlog items. Note, that if a mid-sized item has value 95 or a large epic has value 76 like in the figure below this is a clear message to the Product Owner to start working on that item to bring it to the definition of ready, so that such important items can be handled in a near-term iteration.

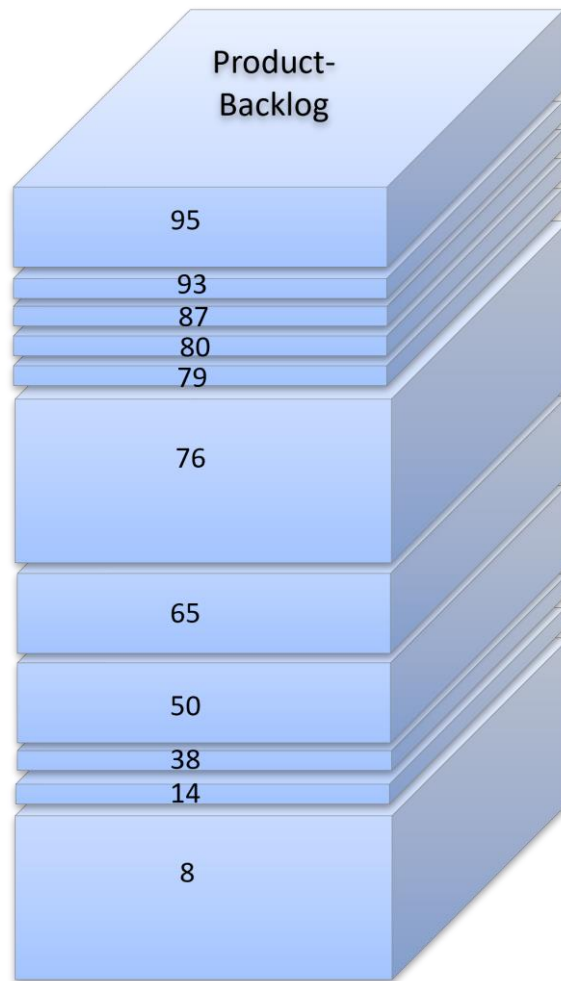


Figure 21: Using a range of numbers to indicate business value

The simplest way would be to sort all backlog items in a linear sequence (that is putting story cards in a row from left to right). The further left the more important the backlog item is considered to be. The further right you put it, the less important this item is considered to be. This is shown in Figure 22.

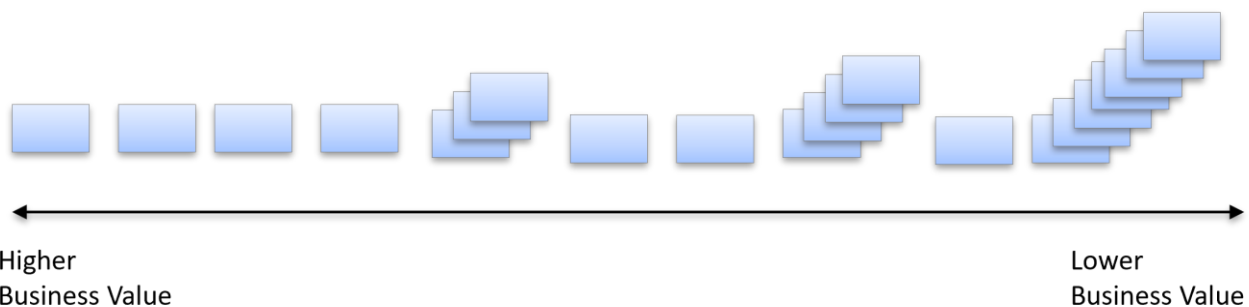


Figure 22: Linear sorting by business value clusters

Note that only the leftmost items have to be clearly linearized since the developers will pick them for the next iteration. The further to the right an item is placed, the less important is its exact position. So, you can put clusters of items on stacks without explicitly deciding their exact value.

The Product Owner has time for refinement before they are picked for implementation. Do the sorting from left to right quickly and only concentrate on those items that promise high business value.

Of course, you could apply much more complex algorithms to determine value. You can for instance pick a couple of criteria mentioned in chapter 5.1 and assign a weight to each of them for balancing the values relative to each other. You can then individually rank product backlog items within each criterion and calculate the resulting value. Figure 23 demonstrates this with three criteria and a ranking from 0 to 5 within each criterion. As you can see Story 3 turns out to be the most valuable one based on that combinatorial approach of revenue, risk and usability.

Backlog Items	Revenue in 2nd Quarter		Minimizing Technical Risk		Improving Usability		Overall Value
	Weight = 5	Revenue Value	Weight = 4	Risk Value	Weight = 2	Usability Value	
Item 1	3	15	0	0	0	0	15
Item 2	0	0	3	12	1	2	14
Item 3	4	20	2	8	2	4	32
Item 4	2	10	2	8	3	6	24
Item 5	0	0	2	8	5	10	18
...							
...							

Figure 23: Calculated business value based on multiple criteria

5.4 Estimating User Stories and other Backlog Items

For the Product Owner this chapter is for information only. He or she is only responsible for determining the order of the backlog items based on value and risk as discussed in the last chapter. It is the task of the developers to come up with estimates for each backlog item. The Product Owner should not influence the estimation process, only be aware of the results.

Even in a perfect agile world, forecasts are useful and valuable (if applied properly) in order to determine how much work can be “done” within a previously specified iteration (time box). No non-estimated element is allowed to enter a sprint in Scrum for two reasons [Cohn2006]:

1. It is not clear if the element can be completed within the sprint and as a result of this the software may not be working at the end of the sprint.
2. Without discussion and estimate, the team will have no reference point (planning vs. actual doing) for future learning with regard to upcoming sprints.

Most people dislike estimating. In many non-agile organizations inaccurate estimates were typically used against you at a later stage. If your estimate was too high, then you might be seen as too defensive or too anxious. If your estimate was too low, then you could be challenged why you didn't see the real efforts behind the work that had to be done.

Agile organizations try to overcome this dislike by establishing a different kind of estimation culture. A culture that helps avoiding finger pointing. The principles of this culture will be discussed in this chapter.

First and foremost, reason for having better estimates is the use of short iterations in agile development. It is much easier to give more precise estimates for the next two to four weeks compared to estimates for quarters or for years.

Of course, development organizations that work on large projects with multiple teams also need forecasts in order to prioritize and plan work properly. Large scale estimating and planning will be discussed in more detail in chapter 6. In this chapter we will concentrate on short-term estimating, for example estimates for the next couple of iterations.

Agile methods suggest some good practices that help to have better and more accurate estimates:

1. Everyone involved in the estimation process must have the same understanding of the work that needs to be “done”. This is achieved by involving the developers in the product backlog refinement. Developers assist the Product Owner in refining unclear epics features and stories or any kind of requirements on those levels of granularity, thereby gaining more insight into the work to be done. Creating such a common understanding of what “done” really means in this context avoids typical estimation pitfalls (forgetting about efforts needed for documentation, testing or rollout preparation).
2. Estimating is done by those doing the work; the cross-functional developers. This helps to bring all involved people on the same level of knowledge by exchanging knowledge and sharing assumptions about the work to be done. Of course, you have to consider a tradeoff between involving all team members in the estimation process and involving only some of them. Involving all means everyone is part of the process and therefore feels committed to the outcome. But this might take a lot of time that could otherwise be spent on developing features. If only a few developers participate in the estimation process, then the others may not feel committed. A good practice is to invite the whole team and let the team decide who is really needed to estimate. In all cases estimating should be done by groups and not by individuals. Later in this chapter we will suggest techniques to speed up estimating.
3. Estimating should be done relative to work already done or, in the beginning, relatively to small work everyone involved can agree on. Estimating by analogy or affinity is likely to be more accurate than absolute estimating. Looking at Figure 24 it is easy to state that the rock on the right is more than twice the size compared to the rock on the left. It would be much harder to estimate the exact size or weight of the two. Relative estimates offer enough precision for planning.



Figure 24: Relative estimates

4. Estimating should be done using an artificial unit (usually called story points) representing effort, complexity and risk in one. Using an artificial unit like story points is necessary to make everyone familiar with the new way of estimating and the associated culture and move away from the traditional behavior.

Several techniques support the relative estimate. The most well-known techniques are T-Shirt sizing or the so-called Planning Poker [Cohn2006].

For all of these techniques it is relevant to first agree on a reference item (or reference story). Let us assume the apple in Figure 25 is the chosen reference. Now you can estimate the size of all other fruits compared to that apple. Are they approximately of the same size? Are they much smaller? Or much bigger?

Relative estimates remove the fear amongst the developers that they have to be exact.

The size indicators of T-shirts range from extra small to extra large (XXS, XS, S, M, L, XL, XXL). Some methods suggest not using all of these size indicators when estimating because they may already be too precise. Think of a subset XS, L and XXL as demonstrated in Figure 25. Of course, a cherry is larger than a blueberry, but both are definitely smaller than apples or oranges. And melons are definitely bigger than oranges, which are similar in size to apples.



Figure 25. Reduced T-Shirt Sizing

In Planning Poker, the developers estimate the backlog items based on a set of cards with numbers inspired by the Fibonacci sequence, representing relative sizing (cf. Figure 26).

If you have agreed on one medium sized reference story, for example 5 story points, the team now decides on the size of other backlog items with respect to the reference story. After everyone has covertly selected a poker card they look at the values: if there are three “5” and two “3” on the table, then the item is marked as a “5” and so on. If the numbers deviate from each other, then the team members with the lowest and the highest estimate discuss the rationale behind their estimates and try to convince the other team members. Then the next estimation round is started. If the team cannot agree on one common value within three rounds, then the requirement is sent back to the Product Owner for clarification.

For the upcoming iterations you may want to be in the range between 1 and 13. A “20”, “40” or “100” is an indication for the Product Owner to refine that item. These numbers do not literally mean “20”, “40” or “100”, but “too large”, “much too large” and “enormous” – but they are at least indicators for “how much too large” compared to the items between 1 and 13. If the team has no understanding of the value, then they should pick the “?” instead of expressing their fear by picking “100”.

Some sets include the “0” usually meaning: “Stop talking; this is not a relevant effort and it is not worthwhile to include in the plan.”



Figure 26: Planning Poker cards

The advantage of Planning Poker is, that it is a very good technique for new and inexperienced teams to find their estimates because it avoids anchoring by single team members. The disadvantage is that it is very time consuming. *[Note: The book “Thinking, Fast and Slow” from D. Kahneman [Kahneman2013] gives a great introduction into anchoring and other psychological effects related to thinking and judgment.]*

T-Shirt sizing or Planning Poker usually takes quite some time, since every backlog item is discussed and estimated individually. To overcome this disadvantage improved techniques can be used by more experienced teams.

One simplification of the Planning Poker technique is based on the same principles as Planning Poker but uses a different way of determining the right estimate. Instead of every team member doing a personal estimate one set of poker cards is spread across a table and the reference requirements are placed in the corresponding “container” represented by the poker card. Afterwards the requirements are selected by the team members in a round-robin approach where the team members are allowed either to place a new requirement in the corresponding “container” or reassign one already placed requirement in a different container. If one requirement is reassigned a number of times, then it will be removed and send back to the Product Owner. This approach is much faster but needs a team that is experienced enough to disagree with assignments done by other team members instead of easily agreeing (“anchoring”).

The next step of evolvement is usually called “Affinity Estimation” or “Wall Estimation”. It is used when estimating larger numbers of requirements for example for rough estimates in preparation of Release Planning. Different to the previous approach, the requirements will not be assigned by round-robin approach, but every team member receives a number of requirements and assigns it silently to the “containers” represented by the poker card set (cf. Figure 27). After the silent assignment, all involved are allowed to inspect the assigned requirements and mark those that are questioned. Usually this leads to a quota of 20-30% requirements that need to be discussed and 70-80% that are accepted by all team members.

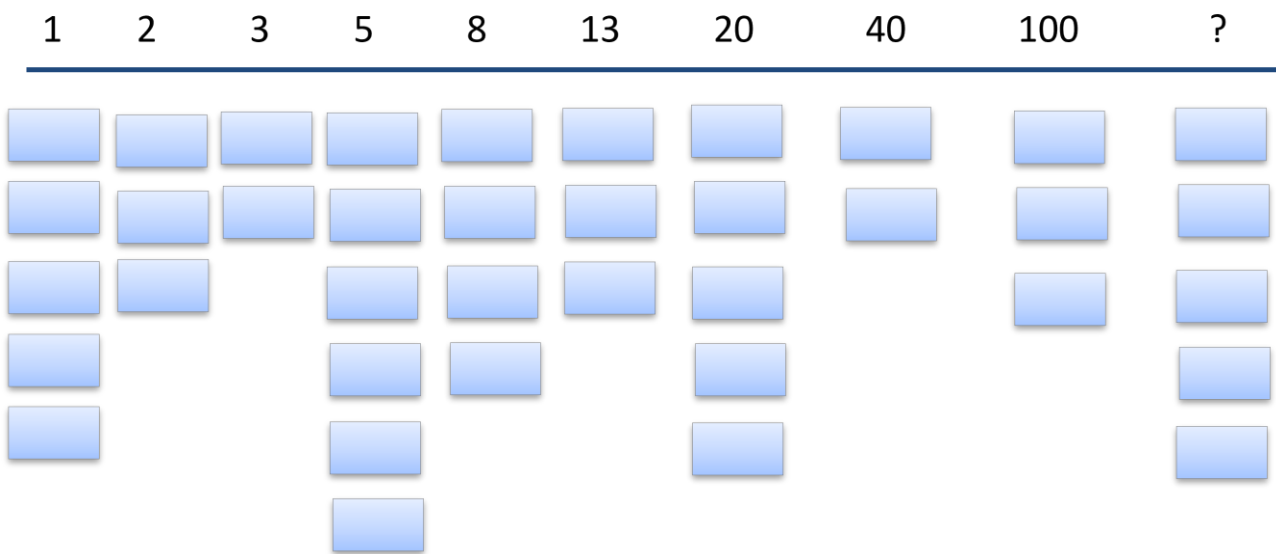


Figure 27: Wall Estimation or Affinity Estimation

Some final remarks about estimating:

Within one team the estimation process will change over time. The team will learn based on the results of finished iterations and it will amend its definition of done to include more precise rules.

While relative estimates have many advantages and work well within one team (as discussed earlier) there are some drawbacks when it comes to estimates across team boundaries. This will be discussed in chapter 6 (Scaling).

Suggestions for Exercise:

Pick a case study and use a quick way to estimate the size of the backlog items. Discuss your findings, especially discuss what did work and what did not work when estimating.

5.5 Choosing a Development Strategy

Different strategies can be applied when selecting what should be picked for early releases, based on known value, risk and effort needed to develop a backlog item. Two concepts are typical for agile development: developing a minimum viable product (MVP) and developing a minimum marketable product (MMP).

Minimum Viable Product

A minimum viable product is the version of a new product that allows a team to collect the maximum amount of validated learning about customers with the least effort. The term was coined by Frank Robinson in 2001 and popularized by Steve Blank, and Eric Ries [Ries2011].

Gathering insights from an MVP is often less expensive than developing a product with more features. Developing a product with more features will increase costs and risks if the product fails, for example, due to incorrect assumptions.

The MVP is a key idea from the Lean Startup methodology developed by Eric Ries, which is based on the Build-Measure-Learn cycle (see Figure 28).

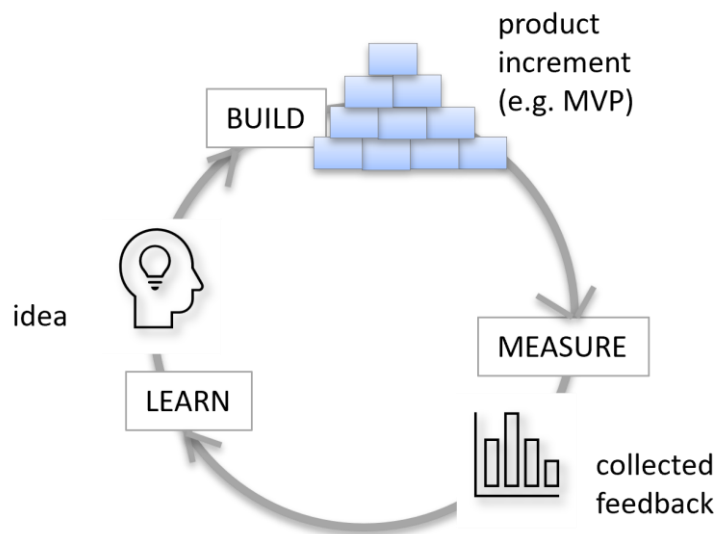


Figure 28: The “Build-Measure-Learn” cycle of lean development

So, an MVP is a learning vehicle, allowing you to test an idea by quickly giving your targeted stakeholders something tangible that allows you to collect data from which you can derive insights into your target market.

Roman Pichler [Pichler2016] observes that “The MVP is called minimum, as you should spend as little time and effort to create it. But this does not mean that it has to be quick and dirty. How long it takes to create an MVP and how feature-rich it should be, depends on your product and market. But try to keep the feature set as small as possible to accelerate learning, and to avoid wasting time and money—your idea may turn out to be wrong!”

The MVP is not necessarily a deployable software product. Sometime paper prototypes and clickable mockups can be used to derive insights as long as they help to test the idea and to acquire the relevant knowledge.

For the iLearnRE system an MVP could be just publishing intro and summary videos for each learning goal to gain insights about user behavior and UI acceptance.

Minimum Marketable Product

The next step should be to create a minimum marketable product (MMP). It is based on the idea that less is more: The MMP describes the product with the smallest possible set of features that addresses the needs of the initial users (innovators and early adopters) and can hence be marketed. Studies have shown that most of our software products contain many features that are never or very seldom used. So, it seems common sense to concentrate on features that are popular for the majority of your stakeholders and delay features that are not considered so popular. To discover these features is not straightforward, but MVPs are an excellent way of achieving this goal. Maybe some of your MVPs are throwaway prototypes created for learning purposes only. But if you do it properly you will develop them in a way that they can be reused or morphed into the first MMP.

If you combine these two concepts you have a strategy that is shown in Figure 29. Develop a couple of MVPs to test the market and get real data as feedback. Then decide on the minimal number of features a product has to have in order to be useful for at least a key group of your stakeholders. Then you continuously add features that promise more business value.

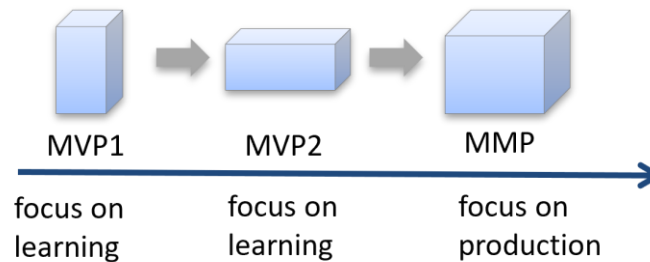


Figure 29: Combining MVP and MMP

Risk Reduction

The development of MVPs is very close to the idea of a risk reduction strategy. Most often MVPs are developed to reduce the risk of having the wrong features for your stakeholders. But you can also create MVPs (or spikes) to reduce technical risks. It is better to fail fast (either in functionality or in technology) than to develop a full-fledged product and then find out it is not successful in the market.

In our iLearnRE case study testing the performance of the planned video platform under load can be a feasible early version.

Low Hanging Fruit or Quick Wins

The opposite of a risk-driven strategy is to go for Low Hanging Fruit first. Begin by publishing features that are easy and quick to implement in order to create early business – earning some money that allows you to invest in more complex features. But beware of postponing risky parts since they may ruin the architecture of a product based on low hanging fruit.

The warning of Professor Kano

Professor Kano conducted studies about customer satisfaction in relation to features delivered. As already included in the CPRE Foundation level syllabus you should be able to distinguish three categories of requirements: basic factors (also known as dissatisfiers), performance factors (also known as satisfiers) and excitement factors (also known as excitors or delighters).

Kano warns that every successful release of a product should include features from all three categories. When you constantly only provide basic factors, your customers will not be very happy. You have to include some performance factors, for instance features that customers explicitly ask for even if they are not absolutely necessary. And you should also try to innovate by including features they did not ask for but will delight them as soon as they receive them.

Creating such a mix of features for each release is difficult to achieve. This is the reason why you should continuously test your markets with MVPs as mentioned above and gather real data before you moving towards time-consuming and expensive feature development.

WSJF

Another interesting strategy for development is the Weighted Shortest Job First (WSJF) approach. It is based on the ratio of the cost of delay and the effort estimated for development [Reinertsen2008].

$$WSJF = \frac{Cost\ of\ Delay}{Duration}$$

Cost of Delay is much more than the benefit (business value) if the respective requirement will be developed. It also includes the perspective what happens if the respective requirement will not be developed (for instance loss of market share, contract penalties) or if the development of that requirement will reduce the risk for the entire implementation (proof of concept) or open up a new opportunity (for instance the use of frameworks which will lower the effort for development in the future).

WSJF can help determine which requirements (or which parts) should be developed first without knowing all details exactly by just using the relations between the requirements regarding Cost of Delay and Duration (development effort).

Backlog Item	Business Value		Time Criticality		RR / OE		CoD		Duration		WSJF
Item 1	8	+	5	+	13	=	26	/	2	=	13
Item 2	1	+	13	+	1	=	15	/	3	=	5,0
Item 3	8	+	1	+	8	=	17	/	2	=	8,5
Item 4	13	+	8	+	5	=	26	/	1	=	26
Item 5	5	+	2	+	2	=	9	/	3	=	3,0

Figure 30: WSJF example

The table is constructed as follows:

- Fill the column with the items/requirements that shall be rated (in our example items 1 - item 5)
- Fill the columns (except CoD) from left to right column per column:
 - Business Value – which value is added if the item is developed?
 - Time Criticality – which value is lost if the item will not be developed?
 - RR (Risk Reduction) / OE (Opportunity Enablement) – how much risk can be reduced or how much opportunities can be taken if the item is being developed?
- Find per column the element that has the LEAST value per column and assign it a “1”
- Rate all other items in the column as a factor in relation to the “1” (you can use any number, but the usage of the Fibonacci sequence is a good practice)
- Calculate the CoD Value as a sum of the previous columns
- Calculate the WSJF as the ratio of CoD / Duration

The item with the highest WSJF ratio should be developed first followed by the item with the second highest ratio and so on.

Using this approach typically means that big chunks will be developed later since big chunks normally have a low ratio. So, the suggestion to the Product Owner is to split big chunks and identify those parts that deliver high value for respectively low effort and further postpone the less valuable parts.

5.6 Summary

Ordering the backlog is an iterative two-step process. As a Product Owner you will preorder the backlog based on business value during the first step. You have seen various ways to define what business value means in your organization. As a Product Owner you should not underestimate risks. Sometimes you have to balance value with risk in order not to endanger your product development. Value can be expressed on various scales like MoSCoW, or High, Medium and Low. Or you simply put all items in a linear sequence based on their value. Then you don't have to use numbers.

Step two is for developers to give you estimates for each backlog item. Agile has done many things to make the estimation process less threatening:

- ▶ The right people (those that do the work) estimate.
- ▶ Estimating is done as a group exercise, not by a single person.
- ▶ Estimating should be done relatively; comparing the size and effort of items instead of giving them an absolute value.

Various processes can be used to estimate, like T-Shirt sizing or using Fibonacci cards in Planning Poker. To speed up the process Wall Estimation or Affinity Estimation can be used.

When the backlog items are small enough and well understood the estimates will be precise enough to allow iteration planning. When the items are still too big or not fully understood the team will indicate that with a higher value – giving a message to the Product Owner that such items need clarification and/or refinement.

As soon as the items are estimated, the Product Owner might change the order of the backlog once more, for instance exchange a group of cheaper items with one more expensive item.

Based on the determined value and the estimates a number of different strategies can be applied to determine the sequence in which items should be assigned to iterations. Strategies like creating a series of minimum viable products (MVPs), followed by a minimum marketable product (MMP) before adding more and more features support the agile principle of deliver early and deliver often. But also harvesting low hanging fruit or reducing risk early on, are feasible alternatives.

An organization may adopt a strategy of early business value gain, for example, if its primary goal is to deliver a product early and establish market share. A strategy of early risk reduction may be preferred if a supplier wants to avoid at all costs that a product is returned due to, for example, inadequate performance or security.

6. Scaling RE@Agile

Requirements Engineering is easier for products that are small enough to be handled by a single team at one location. All the chapters so far implicitly made that assumption: we have shown how the most important requirements (i.e. the ones that deliver the highest business value) can be implemented by that team without the need to distribute requirements among multiple (development) teams. When this assumption no longer holds – that is, we need more than one team to achieve our business goals and visions - we have to consider scaling our development.

In this chapter we discuss why product development must sometimes be scaled and why products have to be developed by more than one team, whether at the same location or distributed geographically. When scaling, the Product Owner of the overall product (as the role responsible for requirements management) is likely to be more challenged with management aspects than with requirements aspects. We will discuss that the two factors *time to market* and *complexity* (either functional complexity or challenging quality requirements) justify and drive the scaling process. But organizational and technical constraints will also influence the way we scale.

In this chapter we will cover the following aspects:

- What does scaling mean and how does it affect requirements and teams (chapter 6.1)?
- How do we (re-)organize the requirements and the teams *in the large* (chapter 6.2)?
- How are releases and roadmaps defined and used in long-term planning (chapter 6.3)?
- How are requirements validated in scaled environments (chapter 6.4)?

6.1 Scaling Requirements and Teams

We use the term *scaling* to describe a change in size, either of the system or the product, or of the number of people involved.

Since around 2010, a number of different agile scaling frameworks have been developed to address these issues. Among them are Nexus [Nexus Guide], SAFe [SAFe1] [SAFe2], LeSS [LeSS], Scrum@Scale [S@S Guide], BOSSA Nova [BOSSANOVA], Scrum of Scrums [SofS], Spotify [Spotify2012], though more exist. Scaling frameworks vary in their maturity level, the number of good practices, guidelines and rules, and the degree of adaptability to the specific needs of an organization. We will not discuss each framework in detail but will rather use them as examples, especially when they present alternative approaches to handling requirements in the large.

In Figure 31 the driving forces for scaling are shown as well as the constraints which may be encountered on the way.

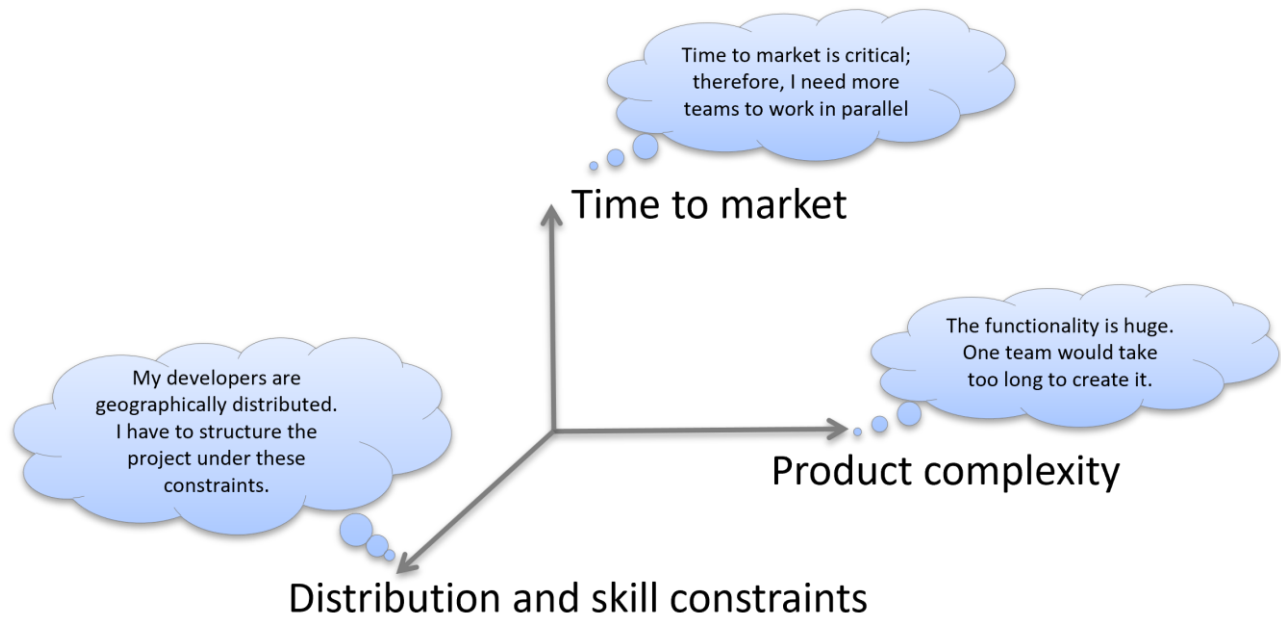


Figure 31: Three dimensions that might trigger scaling

The first two dimensions in the figure above are:

- ▶ **Time-to-market:** One team would take too long to implement all the requirements needed for a satisfactory product. In order to speed up the release you put several teams to work.
- ▶ **Complexity of the product:** The product domain or the technologies used for the implementation are so complex that one team cannot handle all aspects. You therefore decide to work with multiple teams, each focusing on different aspects of the product.

In both cases you are immediately confronted by the fact that you have to coordinate the work of more than one team. This makes development harder compared to working with a single, collocated team.

There is a third dimension shown in the figure above:

- ▶ You might have to work with multiple teams for organizational or political reasons: you may have people in different geographical locations or working across multiple companies, or teams organized around particular specialist skill sets. We consider all of these aspects as constraints that sometimes cannot be avoided, although we wouldn't necessarily recommend choosing these organizational structures where they are not already present. More about good and bad criteria for team structuring in chapter 6.2.

Be careful, however, with scaling when it is not absolutely necessary: working with more than one team always introduces communication and coordination overhead. So, if the reasons for scaling mentioned above do not apply, you probably should not scale at all!

If, however, you do scale, two things will always be true: you will be forced to add hierarchy to the requirements, and hierarchy to the organization. Coarse-grained requirements are needed when discussing the product as a whole; fine-grained requirements will be needed in the teams implementing some aspect of the product. And the teams themselves will need to organize their cooperation to function successfully within a larger team.

How different scaling frameworks tackle these two aspects and what terminology they suggest for hierarchies of requirements and hierarchies of teams is discussed in the following chapters.

6.1.1 Organizing large scale requirements

In chapter 3 we discussed the topic of requirements granularity and introduced the terms *coarse-grained requirements*, *medium-grained requirements* and *fine-grained requirements*. We deliberately chose this more general terminology as the scaling frameworks (and agile requirements tools) differ significantly in the specific terms they use.

Hierarchical representation of requirements reflects one of the key ideas of the product backlog: coarse-grained requirements can still be vague or imprecise until they (or parts of them) become relevant for an upcoming iteration and therefore need more detail and precision. More fine-grained requirements are thus elaborated, and a relationship is maintained to their larger parents. The resulting hierarchy fulfils two purposes:

- ▶ It provides an overview of all known requirements.
- ▶ It allows for the selective detailing of those elements that are most likely to be developed soon.

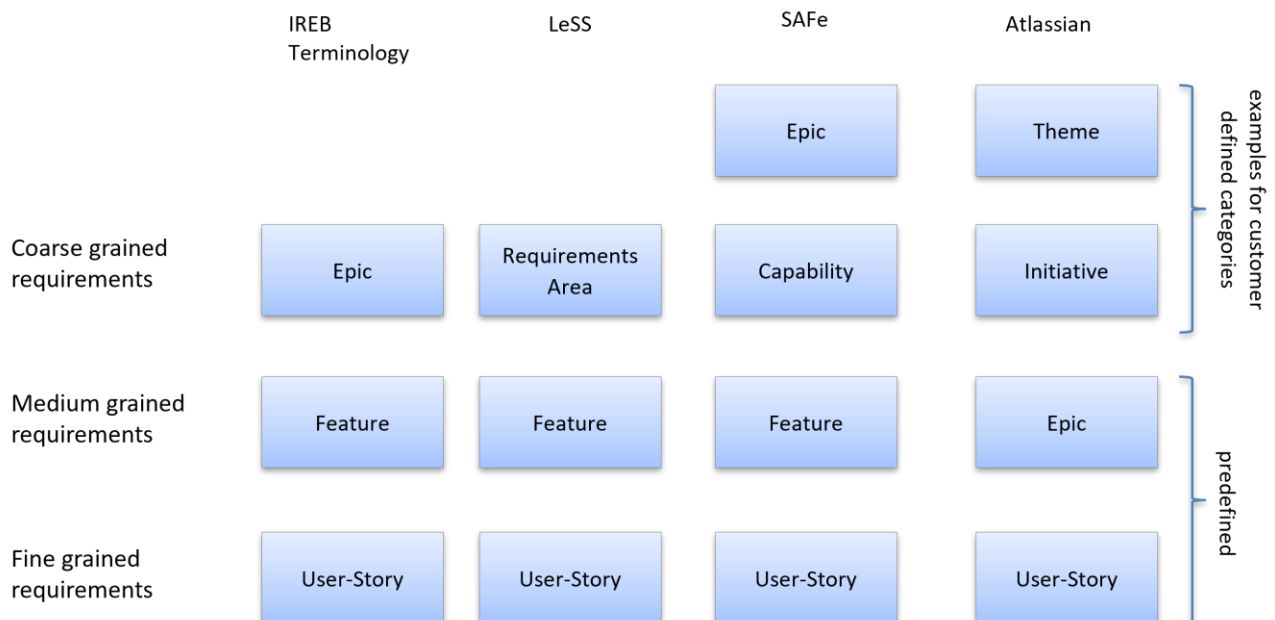


Figure 32: Terminology for requirements at different levels of granularity in selected methods and tools

For the purpose of this handbook, IREB has chosen one of the more popular sets of terms for requirements at different levels of granularity that contains three terms: Epics (for coarse-grained requirements), Features (medium-grained) and User Stories (fine-grained).

Some scaling frameworks and tools do not give explicit names to the distinct levels of requirements, but simply call them *backlog items* and allow their refinement until they are small enough to be implemented in a single iteration.

Other tools start with a two-level approach, but then allow the number of levels to be extended. Atlassian's Jira, for example, uses epics and stories as standard, but allows this hierarchy to be extended (recent versions suggest calling the largest requirements *themes* and the next level *initiatives*). LeSS calls requirements at the level above the user stories *features* and at the largest level *requirements areas*.

The SAFe framework provides an extensive requirements meta-model [SAFeMDM] with four levels of requirements and a strict naming scheme: epics, capabilities, features and user stories. Figure 33 shows a simplified version of this metamodel. The distinction between the levels is not so much based on content, but rather on size.

A story has to be small enough to fit into one iteration (or sprint); a feature must be small enough to fit in one release. Capabilities and epics are so large that they will span more than one release (more about release planning in chapter 6.3).

Note that on each level SAFe distinguishes *business* features - those that create business value - from *enabler* features - the necessary architectural prerequisites without which the business value cannot be achieved. We will discuss this distinction in more detail in chapter 6.2.3.

SAFe also uses specific terms for the acceptance criteria at different levels of granularity, as shown below.

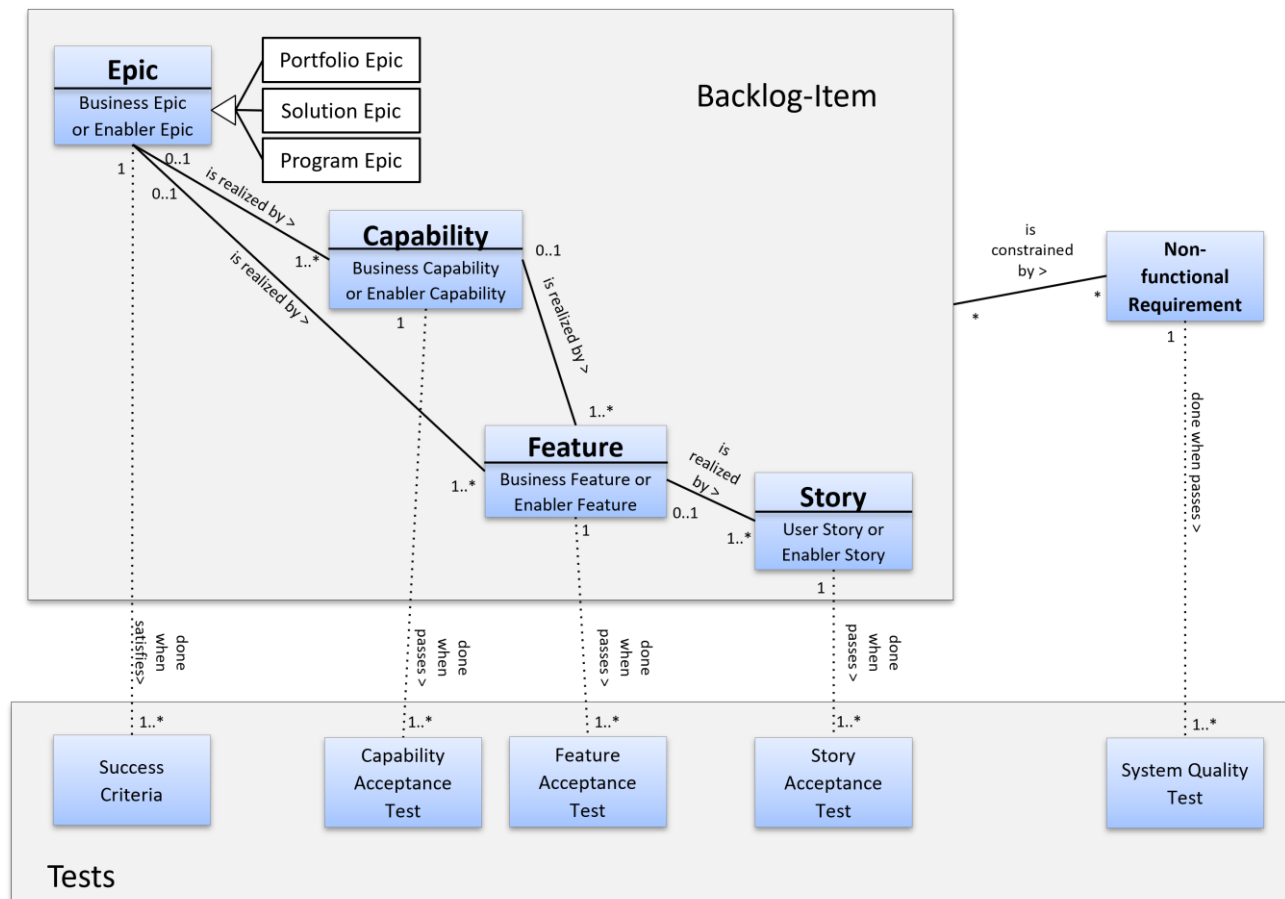


Figure 33: Requirements Terminology of SAFe

Though many of today's agile requirements tools are not capable of handling the four levels of granularity in this meta-model out-of-the-box, most of them provide the means to customize the hierarchy.

In order to avoid lengthy discussions about terminology (and methodology wars among your teams!) we suggest that you decide on an inhouse terminology for the levels of granularity you want to use and then stick to that in every development project. Very often either the scaling framework or the tools you use will dictate the terminology.

6.1.2 Organizing Teams

All scaling frameworks agree that ...

- ... regardless of the specific job titles responsibility is needed at every level in the organization.
- ... work has to be properly coordinated among the teams.

Beyond these general points, however, concepts and terminology differ in specific approaches.

When Scrum is used for multiple teams, one technique often used to coordinate these teams is called *scrum of scrums*. [SofS] The only difference to the work within one team is that each team assigns a person (an ambassador) to represent them in coordination meetings that normally happen two or three times per week. During the course of a project the team can nominate different people, picking the person who can best represent them according to the topics being discussed.

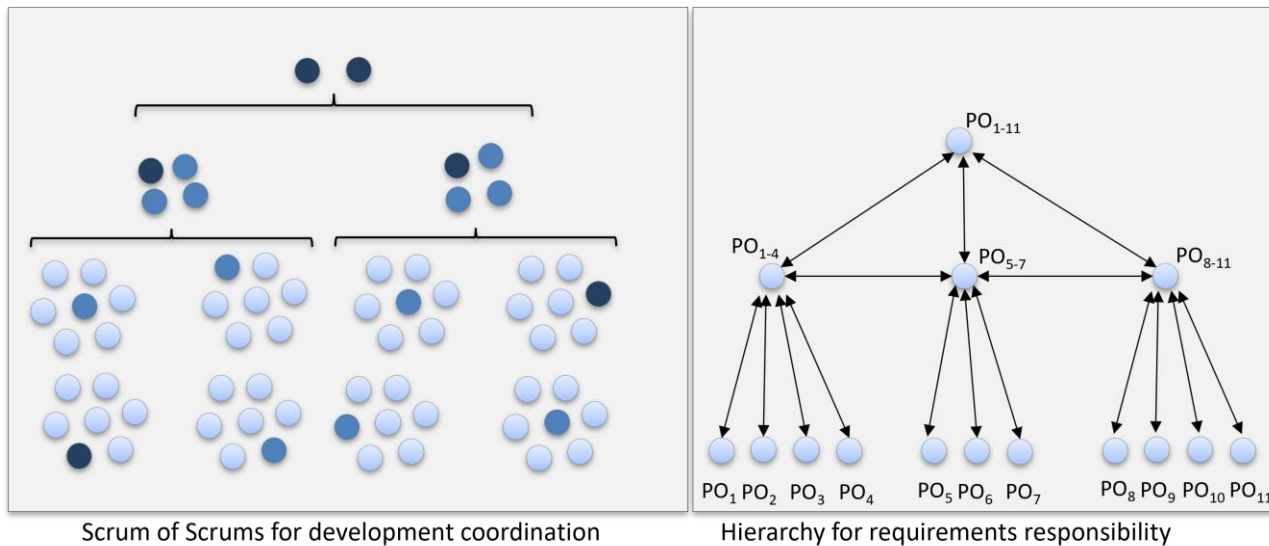


Figure 34: Scrum of Scrums as a model for organizing requirements responsibility

In addition to the general coordination of developers, the requirements hierarchy discussed in chapter 6.1.1 needs a corresponding hierarchy of requirements responsibility (Figure 34, right). Coarse- and medium-grained requirements should be owned by somebody, refinement jobs should be assigned to individual teams and dependencies among the teams should be identified.

The organization of roles at different levels of the organizational hierarchy differs between frameworks: from basic democracy to clearly hierarchical structures.

Among the more democratic approaches are Nexus and BOSSA Nova. They do not suggest having *PO hierarchies*. For those two frameworks the Product Owner is part of the team and the team decides how to coordinate not only the development but also the requirements. Thus, Nexus comes close to the idea of a scrum of scrums (i.e. self-organizing teams) with its *Nexus Integration Team*, which exists to coordinate, coach, and supervise the application of Nexus and the operation of Scrum so the best outcomes are derived. The Nexus Integration Team consists of the Product Owner, a Scrum Master, and Nexus Integration Team members. But note, the Nexus Integration Team is not a decision-making authority: similar to a scrum master of an individual team, the integration team mainly ensures that the required communication takes place amongst the teams in order to solve shared problems.

An even more basic democracy is advocated by BOSSA Nova [BOSSANOVA]. Here, a sociocracy [SOCIOCRACY] is proposed as the ideal form for the organization in the large. The teams select their ambassadors to the coordination circle, and each coordination circle selects their ambassador to higher-level coordination circles, and so on.

Other frameworks establish clearer requirements management structures with well-defined decision-making authority. They often assign fixed job titles to the requirements coordinators on higher levels. As we saw above with requirements hierarchies, the exact terminology used in the organizational hierarchies also varies among the different frameworks. Figure 35. gives an overview of some of the job titles and role names used in selected frameworks.

Framework	Nexus	Scrum@Scale	SAFe	LeSS
Requirements Coordination ...				
... on Portfolio Level (organization wide)			EPIC Owner	
... of Teams of Teams of Teams		EMS (Executive Meta Scrum)	Solution Management	
... of Teams of Teams	Nexus Integration Team	Chief Product Owner	Product Management	Product Owner
... on Team Level	Product Owner	Product Owner	Product Owner	Area Product Owner

Figure 35: Role names for requirements responsibility

Some frameworks (Scrum@Scale, Nexus, SAFe) reserve the role name “Product Owner” for the individual team and propose new role names for the higher-level coordination roles. Scrum@Scale uses the term Chief Product Owner, for example.

In SAFe the *Product Manager* is responsible for the output of multiple teams, who together form an *Agile Release Train*. Where multiple *Agile Release Trains* work together to fulfil the requirements of an even larger solution, they are managed by a *Solution Manager*. At the largest level of granularity, corporate-wide agility, *Epic Owners* have overall requirements’ responsibility and together represent the *Portfolio Management*.

LeSS goes the opposite way and states that even for large teams the responsibility is with the Product Owner. Individual teams can then assign *Area Product Owners* to manage requirements for the part of the product assigned to smaller teams.

You should remember: Job titles do not matter as long as there is someone (or a small group) that is responsible for managing requirements. All frameworks suggest working with a single product backlog, independent of the size of the team (see more details about logical backlogs in chapter 6.2). Parts of that single backlog can then be assigned to sub-teams.

Whatever mechanism you use, make sure that the sub-teams (or their representatives) communicate on a regular basis about overlaps, dependencies and priorities in order to achieve the best outcome for the overall developers.

6.1.3 Organizing Lifecycles/Iterations

In our definition in chapter 1.3 we stated that RE@Agile is an iterative process. For large projects, most of the scaling frameworks suggest two different kinds of iterations:

- ▶ Short iterations (often called sprints): where individual developers try to implement the backlog items allocated in the sprint planning meeting. These short iterations typically last between two and four weeks.

- ▶ Longer iterations (often called releases): mainly intended to ensure integration of the results of multiple teams. Releases can contain a number of short iterations. Different frameworks establish different rules for how frequently to integrate, ranging from integrate in every iteration to integrate at least in every release. Release iterations should not last longer than two to three months.

For more about release planning and roadmapping see chapter 6.3.

6.2 Criteria for structuring Requirements and Teams in the Large

In large-scale product development multiple teams have to work together on the same product. In practice, each team develops a specific product slice that must be integrated with other slices to build a working solution. Only the integrated product has value for the stakeholders.

When scaling product development to multiple teams, it is not sufficient for all Product Owners to simply meet and somehow discuss which teams should develop which part of the product, and then to hope for the best! Sophisticated structures and practices are needed to support team collaboration, manage requirements changes and enable rapid product delivery. Otherwise, developers may waste effort coordinating with teams that are not relevant for their work.

From a requirements perspective we have to close the loop: from the initial (business-) requirement demanded by stakeholders, through the splitting of complex requirements into smaller pieces manageable by developers, and then onto ensuring that the assembled results combine to form a solution that can be released to the business.

6.2.1 Product-focused backlog

Product Owners need a shared understanding of the product and its business context. This is important as they need to work collaboratively on requirements at different abstraction levels and to agree on individual teams' priorities, which should also reflect overall business priorities. Furthermore, agile teams must identify requirement overlaps and dependencies in order to minimize interruptions during development.

To support this kind of product focus, requirements must be managed using one logical backlog. The key idea is that each requirement is held in one place only, avoiding redundancies and contradictions. This can still be achieved even when further sub-dividing the backlog into team backlogs, as illustrated in Figure 36. While refining coarse-grained requirements, Product Owners may work on backlog items not yet associated to any team (see (a) in Figure 36) or they may split complex requirements and hand the resulting backlog items to the teams for further refinement (see (b) and (c) in Figure 36). To ensure traceability among requirements on different abstraction levels, Product Owners should link the backlog items.

For example, considering a complex requirement that describes the connection of a specialized hardware device with a computer app using a proprietary protocol. This requirement is initially stored in the product backlog (see (a) in Figure 36). Assuming, that Team A and B develop the system, whereas Team A has experience with the hardware device. Thus, the complex requirement can be split into a smaller requirement focusing on the interface of the hardware device, which is managed in the backlog of Team A, and another requirement describing the handling of the connection within the app (see (c) in Figure 36), which is managed in the backlog of Team B.

Depending on the tool that is used for backlog management, you can either define team filters on the common product backlog, or you can create (virtual) backlogs for each team. Regardless of the chosen tooling, all backlog items together form one logical backlog.

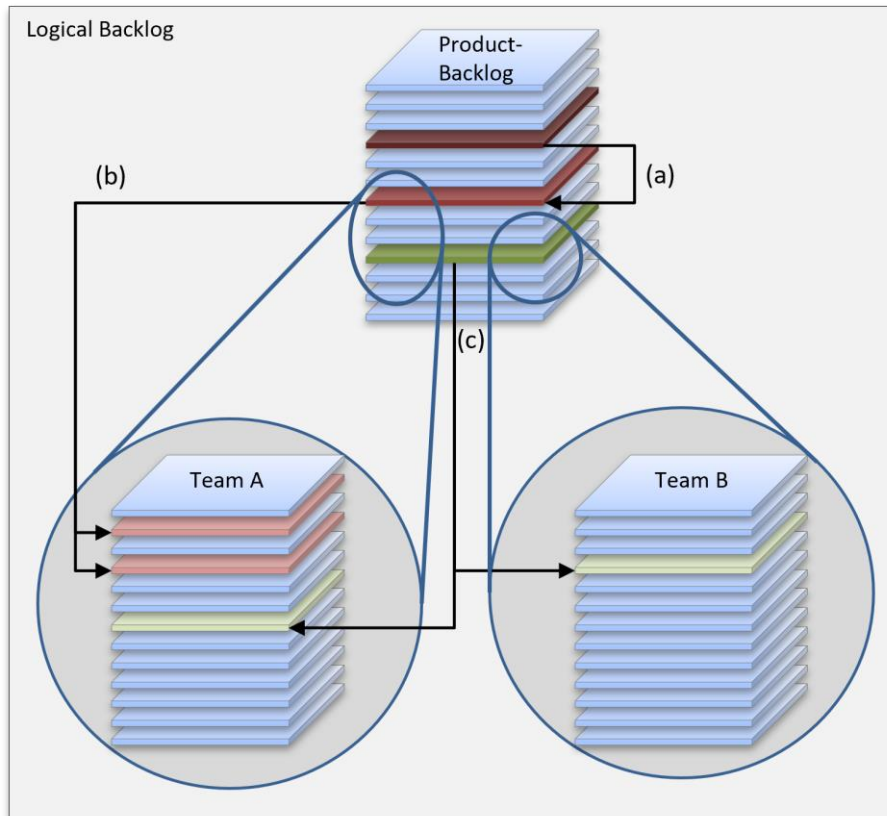


Figure 36: Key idea of the logical backlog approach.

In scaling frameworks such as Nexus, SAFe and Less, one logical product backlog is recommended as well. In SAFe, the logical backlog is split into different backlogs which are linked according to their scaling level (e.g Portfolio Backlog, Solution Backlog, Program Backlog, several Team Backlogs). Each backlog contains requirements of appropriate granularity according to the scaling level. For example, backlog items from the Program Backlog are refined in Team Backlogs, while additional items arising from the team's local context may also be added directly to the Team Backlogs.

6.2.2 Self-organizing teams and collaborative decision-making

Product development will find it hard to react to changes in a timely fashion if each team depends on a complicated web of interactions with other teams to approve any decision. A team structure is required that allows teams to self-organize around value creation: to better respond to stakeholder feedback, to make reasonable decisions independently and to deliver end-to-end features [Anderson2020].

The benefits of self-organizing teams are one of the Agile principles [AgileManifestoPrinciples]. Localized, direct communication within teams (intra-team) allows for optimizations and effective decision making, while communication between different teams (inter-team) is slower and should, in general, be kept to a minimum [Reinertsen2008].

Nevertheless, there will always be a need for collaboration within a network of teams working towards a shared goal. A level of communication and coordination is required that will, inevitably, constrain the level of freedom enjoyed by individual teams.

In order to both work on requirements collaboratively, and to take reasonable decisions autonomously, teams need a general understanding of the requirements of the other teams with whom they have to collaborate, without, though, becoming overwhelmed with all the details. Product Owners should therefore find an appropriate level of detail, sufficient for teams to understand the impact of their decisions on other teams.

6.2.3 Understanding feature-based requirements splitting

Splitting requirements is necessary in agile development to break down larger requirements into more fine-grained ones, which can be implemented in one iteration. As discussed in chapter 3.4, different splitting techniques exist that should be applied in agile development regardless of how many teams are involved. But requirements splitting is much more fundamental in large-scale product development as it enables self-organizing teams which must be able to implement requirements independently from each other.

To deliver shippable product increments with minimal dependencies on other teams, agile teams should work on loosely-coupled, end-to-end features. In our context, the term 'end-to-end feature' refers to a set of coherent functions performing a specific task that provides business value to stakeholders. Depending on the abstraction level at which the splitting is taking place, however, the definition of tasks may range from specific user functions to entire business processes.

To identify end-to-end features, Product Owners must decompose the product scope into units of loosely-coupled and internally consistent functionality (i.e. functional boundaries), as represented in Figure 37. If the scope is split according to these functional boundaries, Product Owners assigned to a particular unit can work on associated requirements with a greater degree of independence. Corresponding teams are often referred to as feature teams [Larman2016].

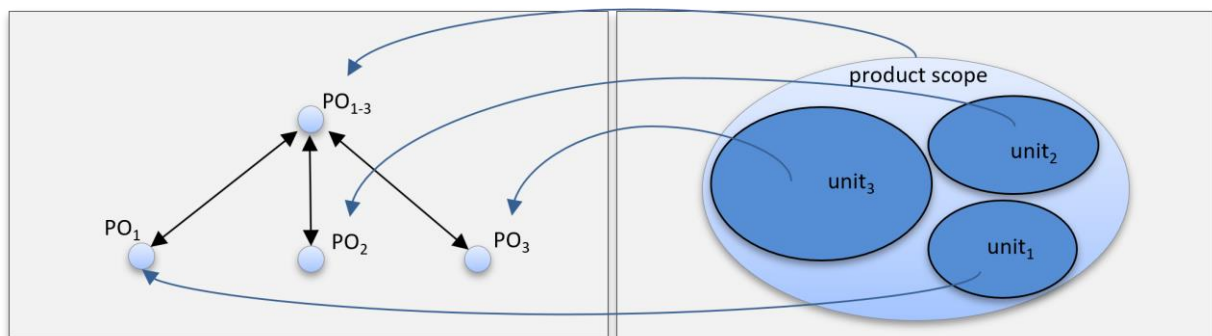


Figure 37: The scope is partitioned to smaller units of end-to-end functionality and shared among Product Owners.

A Product Owner and usually one agile team are assigned to a unit of end-to-end functionality. Boundaries between units help to establish the communication pathways. The boundaries should be clear in order to enable effective collaboration; Product Owners can focus on the detailed requirements assigned to their unit rather than spending a lot of time trying to understand the entire scope and business context. They only have to collaborate with other Product Owners on requirements affecting adjacent units. Requirements can be organized hierarchically based on independent units, as discussed in chapter 6.2.1.

Partitioning the scope of a product can be achieved along business process lines, as discussed in chapter 3.2. If a business process consists of multiple process lines, each line can be supported by end-to-end business-level product features. Ideally, different process lines should be loosely coupled within a business process, which usually allows product owners to work independently on the requirements of their features. In this case, they only have to agree on features that affect the interaction of the process lines.

Use Cases are an approach to structuring requirements, not always typically associated with Agile, but nevertheless recommended by a number of authors (for example Jacobsen, Cockburn, Leffingwell). Use cases view the system as a black box and consider the actions that take place between an actor (human or another system) and the solution.

Use cases may be used as part of the upfront activities to scope and structure a project, as discussed in chapter 2.1.5.3, or elaborated as part of ongoing product development. In contrast to process lines, a use case can be seen at a user-level as an end-to-end functionality of the product. Product Owners must only agree on requirements that relate to several use cases (for example interfaces or common business entities).

6.2.4 Considerations when feature-based requirements splitting is not possible

Unfortunately, in many cases it is not that easy to decompose requirements based around loosely-coupled units of end-to-end functionality. Due to architectural design (for example technology, infrastructure, system components, common platform, architectural layers such as front- and backend) as well as organizational considerations (specialist skills, team location, sub-contractors), units of functionality may overlap as illustrated in Figure 38. This means that different agile teams must work together to implement specific features and their respective Product Owners need to collaborate more closely on requirements [Figure 38]. Alternatively, a dedicated team can be established to specifically work on the overlap, and to collaborate with each of the original teams focused on a unit of functionality.

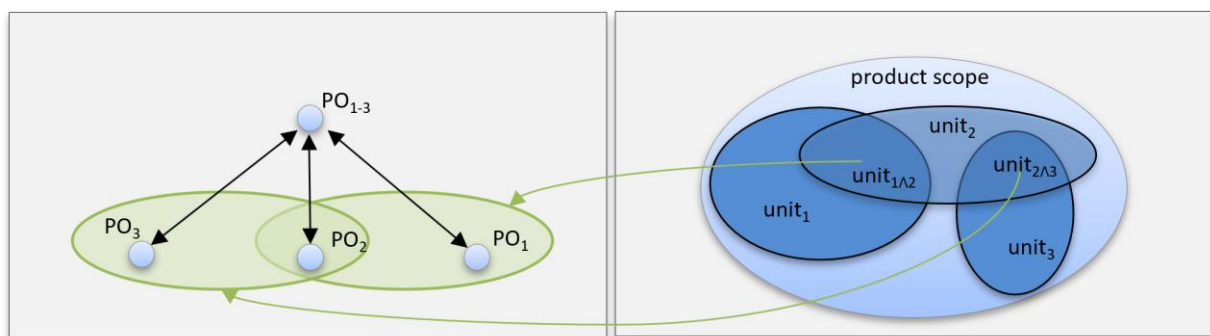


Figure 38: Intersecting units indicate close collaboration of Product Owners with respect to requirements.

To implement features collaboratively, agile teams require a shared understanding of requirements and their business context. They must also agree on overlapping (cross-cutting) requirements, constraints and common technical interfaces so that deliverables from different teams can be integrated to working increments. Integration and testing of features become more complex and synchronizing teams using backlogs and roadmaps is even more critical (see chapter 6.3).

Distributed team locations across different time zones present particular communication challenges and require greater effort to coordinate. If developers from several distributed teams need to implement certain features together, for example, Product Owners must spend more time in decomposing requirements of those features in order to minimize expensive communication. Meetings (virtual or physical!) must be organized explicitly with additional planning effort and at potentially inconvenient times. Different spoken languages or cultures may present further problems.

Teams distributed in different locations but in the same or adjacent time zones do not have all these difficulties, but nevertheless require some effort to organize effective communication, whether through virtual or physical meetings or using other collaboration tools. Video conferencing and collaborative tools can be of much use here.

A special form of distributed teams are sub-contracted teams. Such teams are not necessarily geographically distributed, but rather organizationally distributed i.e., team members are employees of another organization that is in some contractual relationship with other teams.

Ideally Product Owners should not be sub-contracted, as conflicts of interest may prevent them from taking full product responsibility. Sub-contractors often have their own goals, which may at times not fully correlate with the overall product vision or goals.

Each team must deliver value for the product increments. Some teams do not implement features but instead focus on managing infrastructure or helping other teams to integrate deliverables into product increments. For example, SAFe proposes having a dedicated system team which will do the integration of all team artifacts towards one releasable product increment. The Nexus Framework proposes having a "Nexus Integration Team", which is not performing the work but rather providing consultation to the developers on how to do this themselves. Hence, they add value implicitly to the product increment.

Further details on agile organizational design and practices can be found in [Anderson2020].

Finally, we should be aware of the observation of Conway who described a very common pattern known as "Conway's Law". It points out that organizational structure exerts an influence on system design and product structure. In his article [Conway1968], Conway states that organizations that build new systems or products tend to structure their products in the same way that they themselves are currently organized and communicate. The resulting team structure is often sub-optimal with respect to efficient development and delivery in a large-scale agile context.

6.2.5 Telecoms company example

In this example, we illustrate the aforementioned approach for feature-based requirements splitting and discuss the influence of organizational context on the structure of agile teams and their ability to deliver working product features to customers.

Consider the example of a telecoms company looking to develop and launch two new broadband products to their customers:

1. A new high speed VDSL (internet over the telephone line) product "VDSL100"
2. A fibre-to-the-home (internet over optical fibre) product "FTTH1000". In a first phase, Product Owners analysed the two new products and together they established the requirements hierarchy according to the key business processes as shown in Figure 39:

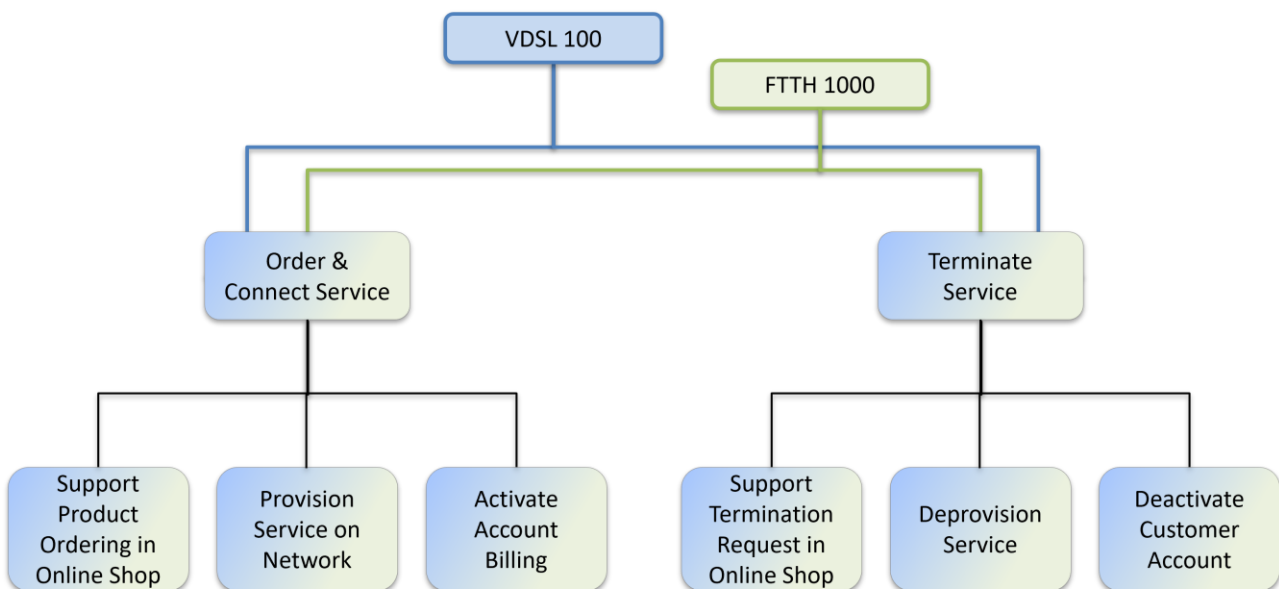


Figure 39: Broadband product requirements structure

Even if the details for each requirement might vary across the two products, the organization of the coarse-grained requirements is the same.

To provide the two products to their customers, the telecoms company must extend its existing IT system. For reasons relating to the organization's history, the key IT systems, as well as the resources and skills within the IT team, are organized as follows (1) *Online Shop and Customer Service Portal*, (2) *Customer Account and Billing System* and (3) *Network Provisioning and Installation Systems*.

That is to say, the online shop and customer services portal is considered a single IT product, with a full technology stack of front-end, business logic and persistence layer. This is also the case for the customer accounts and billing system. Developers typically specialize in one or other of these systems, but not both. The network and provisioning systems are more diverse but are similarly handled by specialist technical roles.

As the organization looks to transition to a scaled agile approach, leaders of the telecoms company meet with Product Owners to discuss the best structure for agile teams. The first proposed team structure and the assigned product requirements are shown in Figure 40:

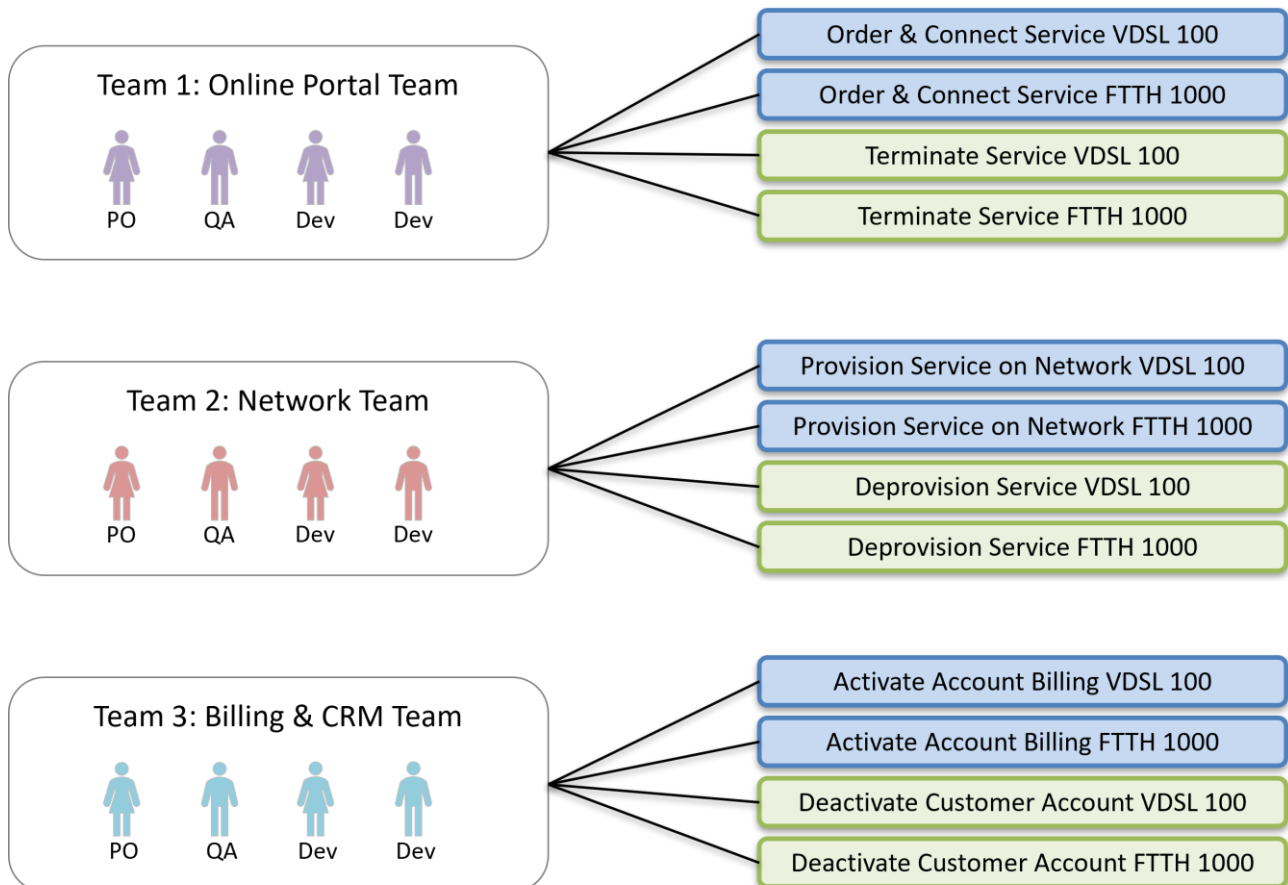


Figure 40: Team structure matching the organizational structure

The composition of the agile teams closely matches the existing organizational structure. The team members are specialists in the corresponding system and work on requirements that address that system. Communication among the teams is primarily required to ensure that the systems work together to successfully launch the two services. No team is able to independently deliver working features fully supporting a customer interested in either product. In addition to each team's Product Owner, who specializes in the requirements of that system, further Product Owners might be required to coordinate the delivery of the coarse-grained, end-to-end process requirements.

To reduce communication effort among teams, a second composition of the agile teams, shown in Figure 41 is then proposed:

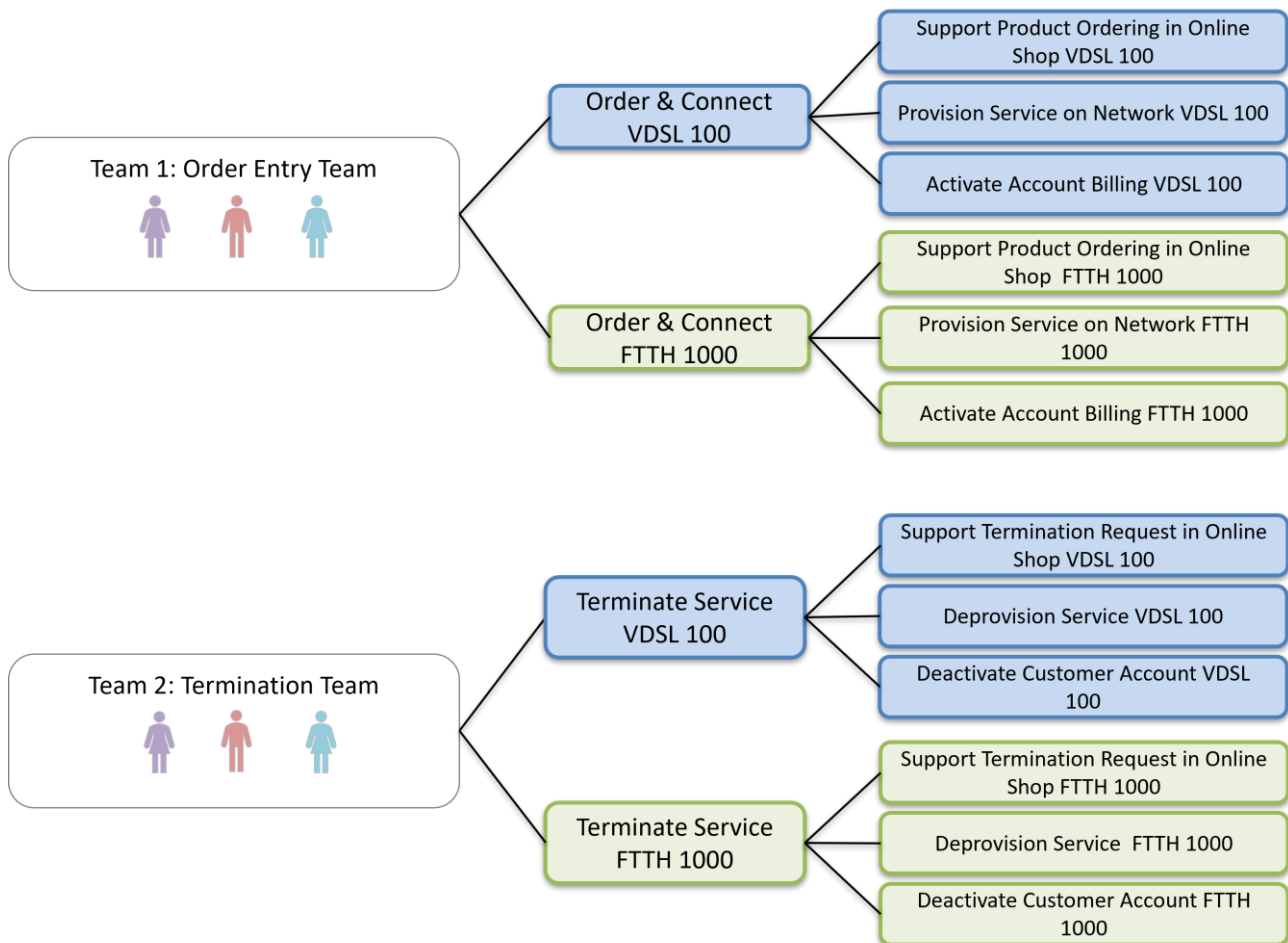


Figure 41: Team structure according to connect and terminate services

Each team is responsible for one key business process and experts from each of the respective systems are mixed in each agile team. Thus, a team is capable of delivering an end-to-end process feature (for example, ordering a broadband product) and providing value to customers (as per the feature teams discussed in chapter 6.2.3). From the requirements point of view, coordination effort is reduced as each Product Owner can design their product with greater autonomy. Coordination is principally required on the product-level (VDSL 100, FTTH 1000), for example to ensure a consistent product model across the different processes. As the integrated solution includes three single IT systems, communication between the teams will be required to coordinate changes and releases within a particular system.

Another composition of agile teams is discussed, shown in Figure 42, also emphasising the concept of teams with full end-to-end capabilities:

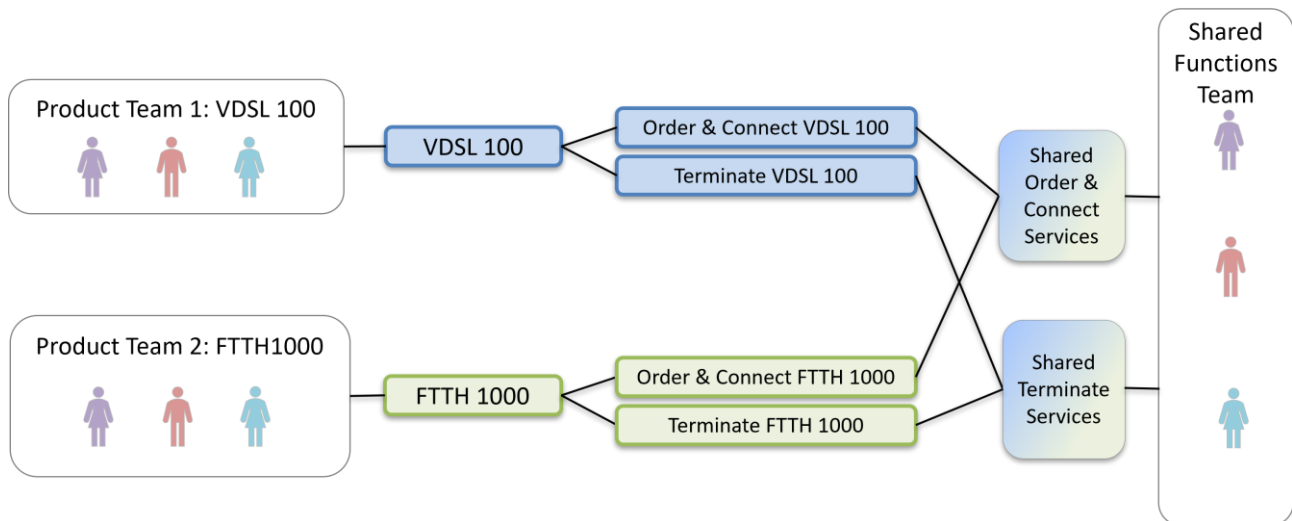


Figure 42: Team structure with full end-to-end capabilities

Here each agile product team is capable of fully delivering a marketable product with all its features (VDSL 100, FTTH 1000). With expertise across all systems and all business processes each team is able to deliver business value independently. From an agile perspective, this team structure should be preferred. In practice, however, these teams run a high risk of duplicating functionality as they work on the overlapping requirements. To address this issue a shared functions team, specializing on just these overlaps, is suggested, and is tasked with finding generic solutions across the two products: leveraging existing systems and services where possible, or developing enabler features where appropriate to support these and other products (see the distinction between business features and enabler features in 6.1.1).

So which approach should we choose? Unfortunately, there is no simple answer. As discussed above, the preferred approach will depend on many factors: the existing organizational structure, its willingness to change, technical and architectural constraints as well as the degree of shared functionality across the different products and processes. Ideally, we would first structure the requirements and then aim to build feature teams as far as possible, but in truth a balance must be sought after careful consideration of all these factors.

6.3 Roadmaps and Large Scale Planning

In large-scale product development, Product Owners manage requirements in the product-focused backlog as discussed in chapter 6.2.1. In contrast to the backlog, a roadmap is used for planning product development incrementally. A roadmap is a prediction of how the product will grow [Pichler2016]. Roadmaps do not change the content of backlog items but arrange them onto a timeline. It answers the question when we can roughly expect which features.

A roadmap is a useful means to communicate (strategic) goals and decisions to the developers and other stakeholders. It breaks down a long-term goal into manageable iterations, represents dependencies among the teams and provides direction and transparency to the stakeholders.

A roadmap is the result of a planning exercise, as shown in Figure 43. The basis for planning is on the one hand the ordered and estimated product backlog and on the other hand the available developers and their capacity.

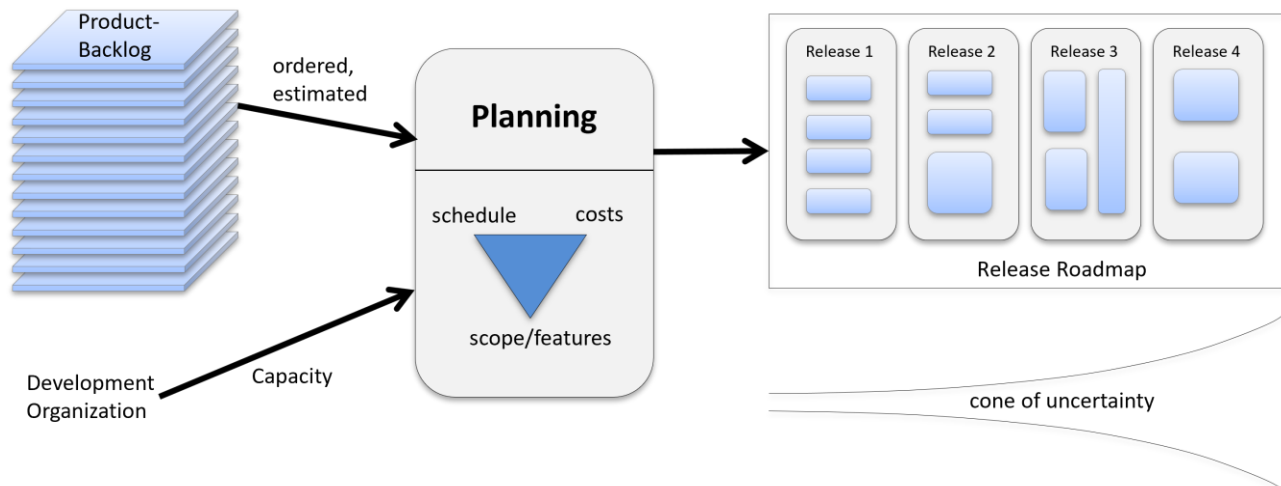


Figure 43: Planning exercise

With this input a Product Owner then faces the typical project management triangle in having to balance scope (features or functionality of the product), costs (available resources) and schedule (delivery dates). We have deliberately drawn the triangle standing on its head to indicate that in agile projects very often costs and schedule are fixed and therefore the planned features are the only variable.

At the beginning of the agile product development, little is known about the product or the work done by the teams. Thus, the scope of the product, as well as the cost estimates, are subject to a high level of uncertainty. As more iterations are completed and as more feedback is gathered from the stakeholders, the uncertainty gradually decreases leading to more reliable planning and a stable roadmap. This principle is known as the *cone of uncertainty* [Boehm 1981]. However, the cone of uncertainty also shows that releases to be published soon, offer greater certainty as to what functionalities will be included, while releases further in the future can only be vaguely defined (see Figure 43). Although this principle is generally true for all agile development projects, it becomes even more important in large-scale product development, as the risks due to product complexity and the potential for misalignment across multiple teams – and consequently the need for more planning – are even greater.

6.3.1 Representing roadmaps

A roadmap shows strategic goals, milestones and coarse-grained requirements (for example feature sets). Important milestones may be either internal or determined by external events such as a trade show or the introduction of new regulation to the market.

The representation of a roadmap depends on its purpose, target group and planning horizon. For customers, management sponsors and the business, a long-term *product roadmap* containing strategic goals and coarse-grained product requirements is often sufficient, with features usually described in business language [Pichler2016].

In SAFe, the product roadmap is called the ‘Solution Roadmap’ and represents long-term milestones, strategic themes and releases. A ‘Solution Roadmap’ typically provides a one- to three-year view, with the level of granularity greater in the near term and then reducing into the long term.

SAFe divides a ‘Solution’ into smaller ‘Program Increments’ which deliver value to the customers in the form of working features. To represent the shorter planning horizon, SAFe introduces the ‘Program Increment Roadmap’, comprising up to four iterations. This offers a more detailed view of the work to be done over coming months.

Another type of *roadmap*, known in SAFe as a ‘Program Board’ [Leffingwell2017], focuses on delivery. This provides developers and their Product Owners with a view of fine-grained backlog times (for example stories or tasks) and the dependencies among them.

A product roadmap of our case study iLearnRE containing strategic goals and coarse-grained features is shown in Figure 44.

You can see here the three next releases: the first one is already committed; the other two are forecasts. Each release is assigned to a pre-defined planning horizon. The features are described in business terms rather than as epics and stories.

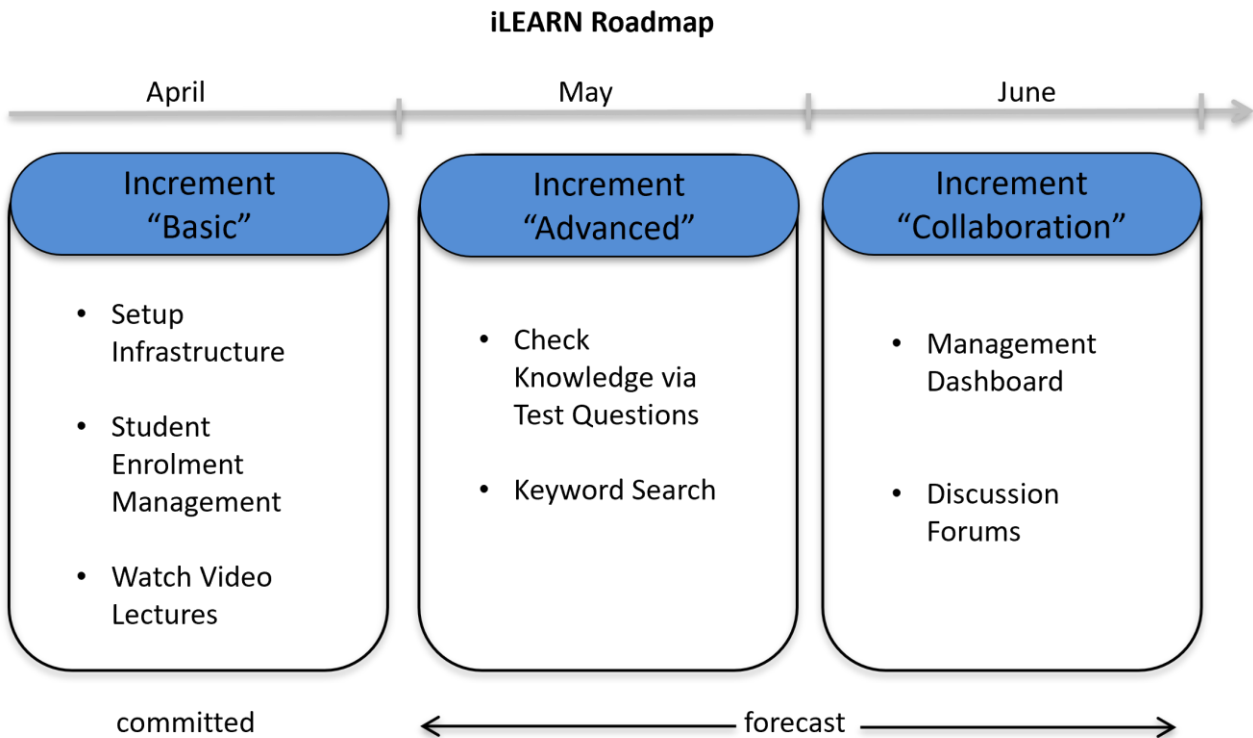


Figure 44: A roadmap for the case study iLearnRE

In chapter 3 we introduced story maps as a way to structure your product backlog. These maps can be extended to display the roadmap for the next releases simply by using the vertical axis to align epics, features and stories to certain releases, thus creating individual release backlogs. This is shown in Figure 45. The items on the story map can be coarser if the release is still some time ahead.

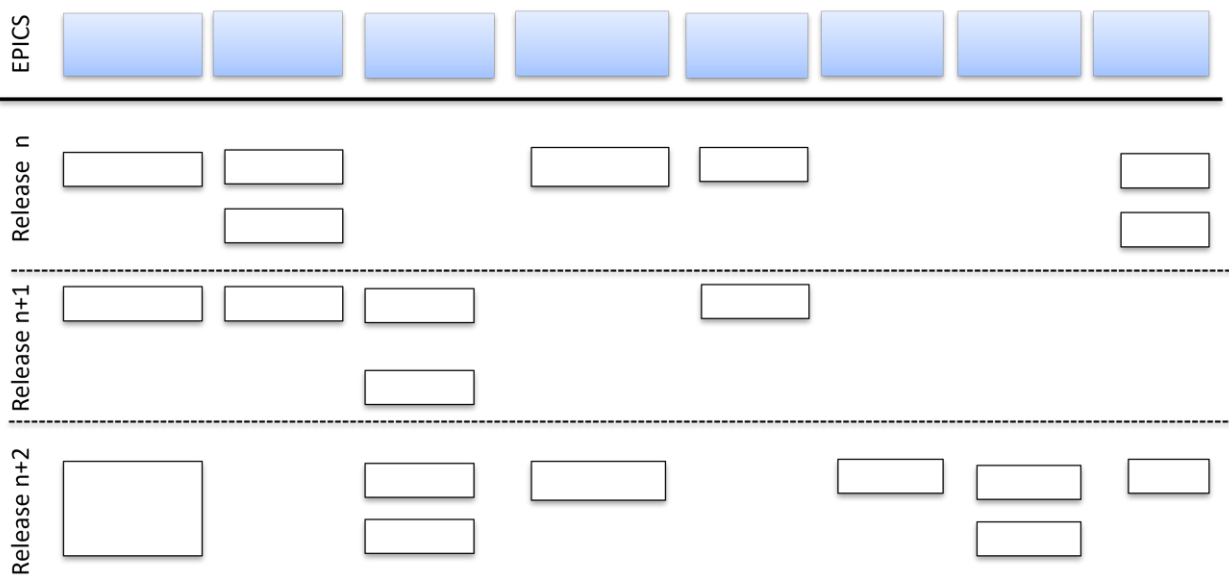


Figure 45: Story maps with release overlay

Using stories and epics to represent the product roadmap has several drawbacks. For most business stakeholders it might be hard to understand how the product as a whole is evolving as too many details are included. Moreover, those roadmaps are prone to changes and must be updated regularly, which is time-consuming.

Figure 46 shows a roadmap which not only includes the planned iterations, but also on the vertical axis an alignment of backlog items to multiple teams, as discussed in chapter 6.2. Program boards are fine-grained delivery roadmaps that are used in SAFe during 'Program Increment Planning'. They contain the language of the developers expressed by backlog items.

The board represents the features to be implemented (F1...F4). The features are broken down into backlog items, here colour-coded. Their order is indicated by the number. The board is used to identify critical cross-team dependencies among work items, as indicated by the arrows.

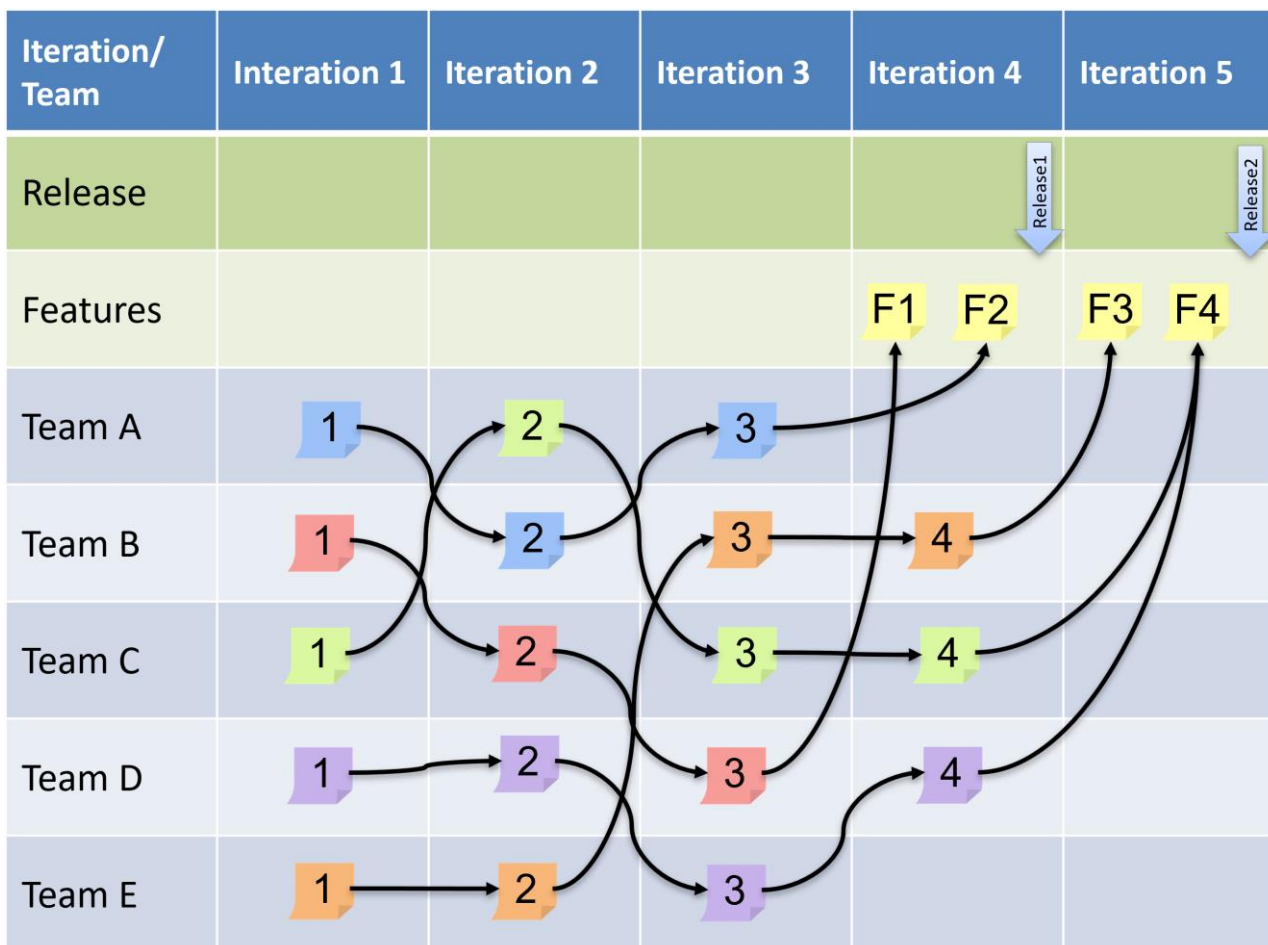


Figure 46: A roadmap with explicit dependencies

If your teams are at the same location, you may be able to maintain your roadmap physically on the wall. If you have to work with distributed teams, you will find dozens of roadmapping tools to support visual planning of multiple releases, many of which are capable to a greater or lesser extent of integrating with the tools used to manage the backlog itself.

In contrast to SAFe, other frameworks such as Less and Nexus do not suggest any specific usage of roadmaps. That does not mean that roadmaps cannot be used within those frameworks, but rather it is up to the developers to decide whether a roadmap is required and which type of roadmap will best support planning and integration work.

6.3.2 Synchronizing teams with roadmaps

Agile development is focused on short iterations with fast feedback cycles, so the ideal situation is one in which the product can be developed with the close collaboration of small groups on a short rhythm.

It is also key that a *regular rhythm* is established for development iterations and releases [DeMarco et al.2008]. Irregular cycles irritate the team, make planning harder and make it harder to track the velocity of the developers.

This rhythm is also called cadence. In music a cadence is a melodic configuration that creates a sense of resolution or finality. For software development this sense of resolution is created on multiple levels of abstraction: within the developers through daily standup meetings, for the developers as a whole in delivering to the Product Owner at the end of a sprint iteration, and potentially for the scaled development organization in creating a shippable product increment for each release cycle.

If you have only one team, delivering a new product increment after every iteration can be done without aligning with other teams. Thus, no other cadence than the iteration cadence (in Scrum the length of the sprint) is needed. If you have multiple teams working on the same product, you need to integrate all team deliverables to a new product increment. As end-to-end testing and the work required to package all deliverables into a release may involve some additional effort, an additional cadence for customer releases may be introduced.

In this sense, a large-scale agile organization can be compared with a large orchestra performing complex music. A well-working, large-scale agile organization shows a kind of harmony. If the organization is not working well, then the harmony is not visible, just like an orchestra that is not playing in time. If you have to work with multiple teams, then the iteration lengths for each team do not have to be identical, but the cycles should be compatible in the sense that they can be synchronized at the level of the larger cadence. Thus, for example, individual teams may choose a sprint length of two or four weeks within a four- (or eight-) week release cycle (see Figure 47).

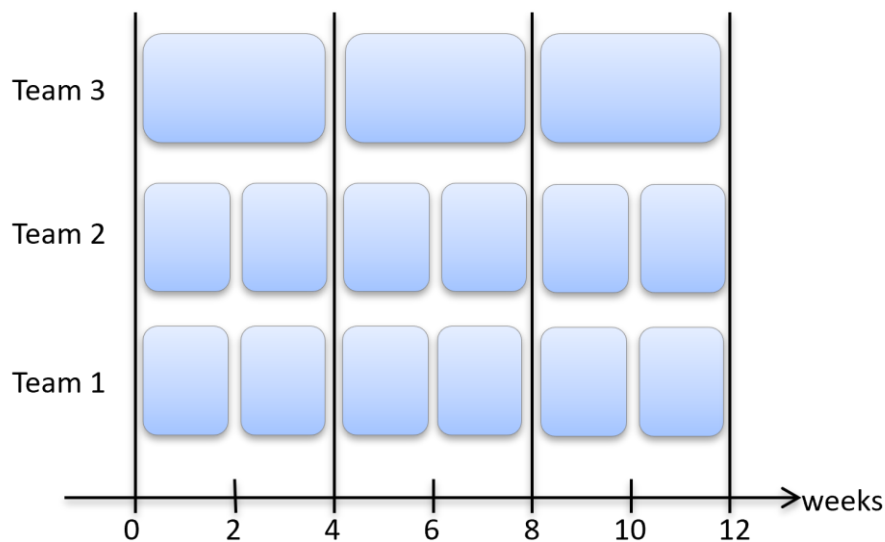


Figure 47: Different but compatible iteration lengths

Manual integration and testing are likely to lead to longer release cycles. Automation can help to shorten release cycles: continuous integration approaches and continuous deployment capabilities may allow teams to deploy features on shorter cycles.

6.3.3 Developing roadmaps

In large-scale product development, requirements work is carried out by different Product Owner roles based around a hierarchy of requirements, as discussed in chapter 6.2.1. Responsibilities with respect to roadmaps will also be different at each level of the hierarchy. On a higher level, for example, Product Owners may be responsible for the product roadmap and, on a lower level, they may be more focused on the delivery roadmap.

To develop a long-term *product roadmap*, a Product Owner must first define a product vision and strategy (see chapter 2). This is necessary so that the right stakeholders are engaged to work on the product roadmap (stakeholder management).

After establishing a product vision and strategy, Product Owners must then elicit coarse-grained requirements (see chapter 3) by engaging with the necessary stakeholders. There is no need to invest time on detailed requirements at this point. Later, during backlog refinement, more details will be discovered.

To gain full support for product development, various stakeholders must be involved early and should understand the business goals of the product. The product roadmap should therefore be tailored to their particular interests and information needs and should be shared and validated with them regularly. Common stakeholders are, for example, executives and senior management, sales and marketing, as well as developers.

Product Owners assign coarse-grained requirements over a broad planning horizon, while also showing strategic goals on the timeline. In an initial product roadmap, Product Owners should avoid hard deadlines. Instead, the features should be planned at the monthly or quarterly level. As product development matures, concrete dates and deadlines can be added.

To create a mid-term *delivery roadmap*, Product Owners must refine the backlog items from the existing product roadmap. These items need to be roughly estimated by the developers, even if the estimates are still imprecise (for example T-shirt sizes) at this stage. The estimate must only be good enough to provide an overview of upcoming iterations.

Experience from practice shows that in most large-scale estimates, the errors for each individual estimate cancel each other out, meaning that the overall estimate is reasonably accurate, despite individual errors.

In chapter 3 we discussed estimation techniques for backlog items. You can also apply the same techniques for longer term estimation and planning. This estimation work is beyond the scope of traditional Requirements Engineering but becomes important in RE@Agile contexts because requirements work goes hand-in-hand with planning. Much more on that topic can be found in [Cohn2006].

Creating and updating *delivery roadmaps* typically happens at face-to-face planning events known as big room plannings (or PI Planning in SAFe), held at regular intervals. In such events, developers collaboratively plan, estimate and prioritize features. Product owners prepare the backlog items upfront and align them to the vision as well as to the existing product roadmap. Teams work with each other to identify the important risks and dependencies. The *delivery roadmap* is updated to show the refined backlog items, the dependencies among them and how they align with the product vision.

6.3.4 Validating roadmaps

The *product roadmap* should also be reviewed from the perspective of the business: customer feedback, market changes, upcoming ideas and markets trends, as well as similar products entering the market, should all be considered. For this purpose, the MMP (as introduced in chapter 5.5) is a good starting point. The validation intervals depend on the stability of the market: in a highly dynamic market, for example, the product roadmap should be reviewed at least monthly, otherwise, quarterly intervals may be sufficient. The key stakeholders should be kept involved with the developing roadmap to increase acceptance and to communicate changes.

In order to narrow the cone of uncertainty, *delivery roadmaps* should also be updated regularly, based either on stakeholder feedback on integrated product increments (see MVP in chapter 5.5), or on the results of prototypes. The validation intervals depend on the maturity of the product development and on changes to the product roadmap. In a mature development process, for example, where senior developers have been working together for some time on the same product, the delivery roadmap may only need to be reviewed after a release. At the beginning of the product development, the delivery roadmap should be validated after integrating the first product increment. Validation of delivery roadmaps can be included within the regular planning events described above.

6.4 Product Validation

A key idea of agile development is to develop a small slice of the product, generate feedback by involving stakeholders and adapt the product development according to the findings and insights gained. Thus, following the principle of the Build-Measure-Learn cycle [Ries2011], product validation becomes an important step to gain rapid feedback. Each time a new product version is released, Product Owners use that product increment to verify its business value and to examine whether the product requirements had been correctly understood.

Product-level validation is an important method in large-scale product development as it ensures that Product Owners together share full accountability from business requirements to product integration. It is the whole product that has value for the stakeholders, not only small product slices.

In Scrum, a sprint review is a suitable measure to present a product increment to the relevant stakeholders. In large-scale product development, a similar idea can be used: but instead of reviewing a single product slice developed by one team, all team deliverables are integrated to a working product increment worth validating. The product increment is demonstrated in a *product review (demonstration)*, showcasing end-to-end features. Thus, stakeholders get a better impression of the entire product [SAFe1], [Larman2016], [LeSS].

To coordinate the integration work that is the basis for product-level validation, a delivery roadmap showing release milestones can be used to synchronize the teams (see chapter 6.2.3).

The challenges of large-scale product development (as mentioned in chapter 6.1) must be considered in product-level validation as well. This means that you must involve a high number of stakeholders and users effectively and communicate their feedback back to the developers. Moreover, you must reach an overall understanding of the integrated product by considering different stakeholder perspectives and knowledge.

When involving many people in a large product review, it is very important to find the right level of detail in discussions to keep all participants interested. One approach is to use a diverge-and-converge collaboration pattern [Design Council]. In the diverge part of the review, the room is divided in multiple areas where teams demonstrate different features of the product increment. As on a bazaar, people walk around, attend demonstrations of interest and give feedback to the corresponding team. Afterwards, in the converge part of the review, people get together to summarize their findings and discuss important aspects and share new ideas.

Product reviews feature in several scaling frameworks. In Nexus and Less the review meeting is called a *Sprint Review*. In SAFe it is known as *System Demo*. According to the Nexus guide, the review should be time-boxed using, as a rule of thumb, roughly four hours for a one-month sprint.

Another approach for product validation in large-scale product development is one that is based on *data analysis* [Maalej et al.2016]. The integrated product increment is delivered to users and, based on their behavior, measurements are made as to whether the product features have a positive, neutral or negative impact. Data analysis frameworks are typically used to analyze feedback data systematically. For example, Product Owners can use the results to identify potentially poorly-designed features. To better understand the identified problems, they may need to again apply regular requirements elicitation and analysis techniques.

However stakeholder feedback has been gathered, Product Owners adapt and re-prioritize existing backlog items and add new items wherever necessary. Some items may be removed from the backlog if it has been shown in product validation that the corresponding features do not generate the intended value. Changes to the product backlog may, in turn, trigger changes to the product and delivery roadmap, as discussed in chapter 6.3.4.

List of Abbreviations

DSDM	Dynamic Systems Development Method
DoD	Definition of Done
DoR	Definition of Ready
LeSS	Large Scale Scrum (https://less.works)
MMP	Minimum Marketable Product
MVP	Minimum Viable Product
PO	Product Owner
RE	Requirements Engineering
ROI	Return on Investment
SAFe	Scaled Agile Framework (www.scaledagileframework.com)
WSJF	Weighted Shortest Job First

References

- [AgileAlliance] Glossary of the Agile Alliance: Definition of term “Definition of Ready”: <https://www.agilealliance.org/glossary/definition-of-ready>, Last visited May 2021.
- [AgileManifestoPrinciples] <https://agilemanifesto.org/principles.html>, Last visited May 2021
- [Alexander2005] Alexander, I. F.: A Taxonomy of Stakeholders – Human Roles in System Development. International Journal of Technology and Human Interaction, Vol 1, 1, 2005, pages 23-59.
- [AmLi2012] Ambler, S., Lines, M.: Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise, IBM Press, 2012
- [Anderson2020] Anderson, J.: Agile Organizational Design – Growing Self-Organizing Structure at Scale. Leanpub, 2020.
- [Beck2002] Beck, K.: Test Driven Development: By Example. Addison-Wesley 2002.
- [BiKo2018] Bittner, K.; Kong, P.; West, D.: The Nexus Framework for Scaling Scrum, Addison Wesley, 2018
- [Boehm 1981] Boehm Barry W.: Software Engineering Economics Published 1981 by Prentice Hall
- [BOSSANOVA] <https://www.agilebossanova.com/#bossanova>, last visited May 2021
- [ClBa1994] Clegg, D.; Barker, R. (2004-11-09). Case Method Fast-Track: A RAD Approach. Addison-Wesley.
- [Clements et al.2001] P. Clements et al.: Evaluating Software Architectures, SEI Series in Software Engineering, 2001
- [Cohn2004] Cohn, M.: User Stories Applied For Agile Software Development, Addison-Wesley, 2004
- [Cohn2006] Cohn, M.: Agile Estimation and Planning, Addison Wesley, 2006
- [Conway1968] Conway, Melvin E.: How Do Committees Invent? Datamation Magazine, 1968. http://www.melconway.com/Home/Committees_Paper.html. Last visited May 2021.
- [Cooper2004] Cooper, A.: The Inmates are Running the Asylum: Why High Tech Products Drive Us Crazy and How to Restore the Sanity
- [DeMaLi2003] DeMarco, T.; Lister, T.: Waltzing with Bears – Managing Risks on Software Projects, Dorset House, 2003
- [DeMarco et al.2008]: DeMarco, T.; Hruschka, P. Lister, T.; McMenemy, S.; Robertson, J+S.: Adrenaline Junkies and Template Zombies – Understanding Patterns of Project Behavior, Chapter 31: Rhythm, Dorset House, 2008
- [Design Council] A study of the design process; [https://www.designcouncil.org.uk/sites/default/files/asset/document/ElevenLessons_Design_Council%20\(2\).pdf](https://www.designcouncil.org.uk/sites/default/files/asset/document/ElevenLessons_Design_Council%20(2).pdf). Last visited May 2021
- [Doran1981] Doran, G. T: There's a S.M.A.R.T. way to write management's goals and objectives, Management Review. AMA FORUM. 70 (11): 35–36 1981.
- [Glinz2014] Glinz, M.: A Glossary of Requirements Engineering Terminology. Standard Glossary for the Certified Professional for Requirements Engineering (CPRE) Studies and Exam, Version 2.0, 2020. <https://www.ireb.org/en/downloads/#cpre-glossary>. Last visited May 2021.
- [HaHP 2000] Hatley, D., Hruschka, P., Pirbhai, I.: Process for System Architecture and Requirements Engineering, Dorset House, N.Y. 2000
- [HeHe2010] Heath, C., Heath, D.: Switch: How to Change Things When Change Is Hard. Crown Business, 2010

- [Highsmith2001] Highsmith, J.: Design the Box. *Agile Project Management E-Mail Advisor 2001*, <http://www.joelonsoftware.com/articles/JimHighsmithonProductVisi.html>. Last visited May 2021.
- [Hruschka2017] <http://www.b-agile.de/Resources/Story-Splitting>. Last visited May 2021.
- [ISO25000] ISO/IEC 25000:2014: Systems and software engineering: System and Software Quality Requirements and Evaluation (SQuaRE) - Guide to SQuaRE: <https://www.iso.org/standard/64764.html>. Last visited October 2018.
- [ISO25010] ISO/IEC 25010:2011: Systems and software engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - System and software quality models: <https://www.iso.org/standard/35733.html>. Last visited May 2021.
- [ISO25012] ISO/IEC 25012:2008: Software engineering - Software product Quality Requirements and Evaluation (SQuaRE) - Data quality model: <https://www.iso.org/standard/35736.html>. Last visited May 2021.
- [Jacobson1992] Jacobson, I. Object-oriented Software Engineering - A Use-Case Driven Approach, ACM Press, 1992
- [Jacobson2011] <https://www.ivarjacobson.com/publications/white-papers/use-case-ebook>. Last visited May 2021
- [HaCh1993] Hammer, M., Champy, J.: Re-Engineering the corporation. Harper, 1993
- [Jeffries2001] Jeffries, R.: Essential XP: Card, Conversation, Confirmation, 2001, <https://ronjeffries.com/xprog/articles/expcardconversationconfirmation/> Last visited May 2021.
- [Kahneman2013] Kahneman D.: Thinking, Fast and Slow. Farrar, Straus and Giroux, 2013.
- [KnLe2017] Knaster, R.; Leffingwell, D.: SAFe 4.0 Distilled, Addison Wesley, 2017
- [Kniberg] Kniberg, H.: Scaling Agile @ Spotify with Henrik Kniberg <https://www.youtube.com/watch?reload=9&v=jyZEikKWhAU&feature=youtu.be>, and <https://www.youtube.com/watch?v=4GK1NDTWbkY&t=156s>. Last visited May 2021
- [Larman2016] Larman, C: Large-Scale Scrum: More with LeSS, Addison Wesley, 2016
- [Lawrence1] Lawrence, R: How to Split a User Story <http://agileforall.com/resources/how-to-split-a-user-story>. Last visited May 2021
- [Lawrence2] Lawrence, R: Why Most People Split Workflows Wrong <http://agileforall.com/why-most-people-split-workflows-wrong/>. Last visited May 2021
- [Leffingwell2007] Leffingwell, D.: Scaling Software Agility – Best Practices for Large Enterprises, Addison Wesley, 2007
- [Leffingwell2010] Leffingwell, D.: Agile Software Requirements – Lean Requirements Practices for Teams, Programs, and the Enterprise, Addison Wesley, 2010
- [Leffingwell2017] Leffingwell, D. et al.: SAFe Reference Guide, Scaled Agile, Inc. 2017
- [LeSS] Large-Scale Scrum: <https://less.works> Last visited May 2021
- [Maalej et al.2016] Maalej, W., Nayebi, M., Johann T., Ruhe, G.: Toward Data-Driven Requirements Engineering. IEEE Software (Volume 33, Issue 1), 2016
- [MaKo2016] Maher, R., Kong, P.: Cross-Team Refinement in Nexus, <https://www.scrum.org/resources/cross-team-refinement-nexus>. Last visited in May 2021
- [McPa1984] McMenamin, S., Palmer, J: Essential Systems Analysis, Yourdon Press, 1984
- [Meyer2014] Meyer, B.: Agile! The Good, the Hype and the Ugly, Springer, 2014.
- [Nexus Guide] <https://www.scrum.org/resources/nexus-guide>. Last visited May 2021

- [OsPi2010] Osterwald, A., Pigneur, Y.: Business Model Generation: A Handbook for Visionaries, Game Changers, and Challengers. John Wiley and Sons, 2010
- [Patton2014] Patton, J.: User Story Mapping – Discover the Whole Story, Build the Right Product, O’Reilly, 2014
- [Pichler2016]: Pichler, R.: Strategize – Product Strategy and Product Roadmap Practices for the Digital Age, Pichler Consulting 2016.
- [Primer2017] CPRE RE@Agile Primer <https://www.ireb.org/en/downloads/tag:re-agile-primer>. Last visited Mai 2021
- [Reinertsen2008] Reinertsen, D.: Principles of Product Development Flow: Second Generation Lean Product Development. Celeritas Publishing 2008
- [Ries2011] Ries, E.: The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses, Crown Business, New York, NY 2011
- [RoRo2013] Robertson S. Robertson J.: Mastering the Requirements Process – Getting Requirements Right, 3rd edition, Addison Wesley, 2013
- [RoRo2017] Robertson S. Robertson J.: Volere Requirements Specification Template, <https://www.volere.org/requirements-auditing-is-the-specification-fit-for-its-purpose/> Last visited May 2021
- [Robertson2003] Robertson, S.: Stakeholders, Goals, Scope: The Foundation for Requirements and Business Models, 2003, <https://www.volere.org/wp-content/uploads/2018/12/StkGoalsScope.pdf>. Last visited June 2021
- [SAFe1] <https://www.scaledagileframework.com/roadmap/>. Last visited May 2021
- [SAFe2] <https://www.scaledagileframework.com/pi-planning/>. Last visited May 2021
- [SAFeMDM] <https://www.scaledagileframework.com/safe-requirements-model/>. Last visited May 2021
- [S@S Guide] Sutherland, J. and Scrum, Inc: Scrum@Scale Guide: <https://www.scrumatscale.com/scrum-at-scale-guide/>, last visited May 2021
- [SOCIOCRACY] <https://sociocracy30.org>. Last visited February 2022
- [SofS] <https://scrumguide.de/scrum-of-scrums/>. Last visited May 2021
- [Spotify2012] <https://blog.crisp.se/wp-content/uploads/2012/11/SpotifyScaling.pdf>
- [Wake2003] Wake, B: INVEST in Good Stories, and SMART Tasks, 2003, <https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/>. Last visited May 2021
- [WyHT2017] Wynne, M., Hellesøy, A., Tooke, S.: The Cucumber Book - Behaviour-Driven Development for Testers and Developers, The Pragmatic Programmers, 2017
- [Yakima 2016] Yakima, A.: The Rollout – A Novel about Leadership and Building a Lean-Agile Enterprise with SAFe