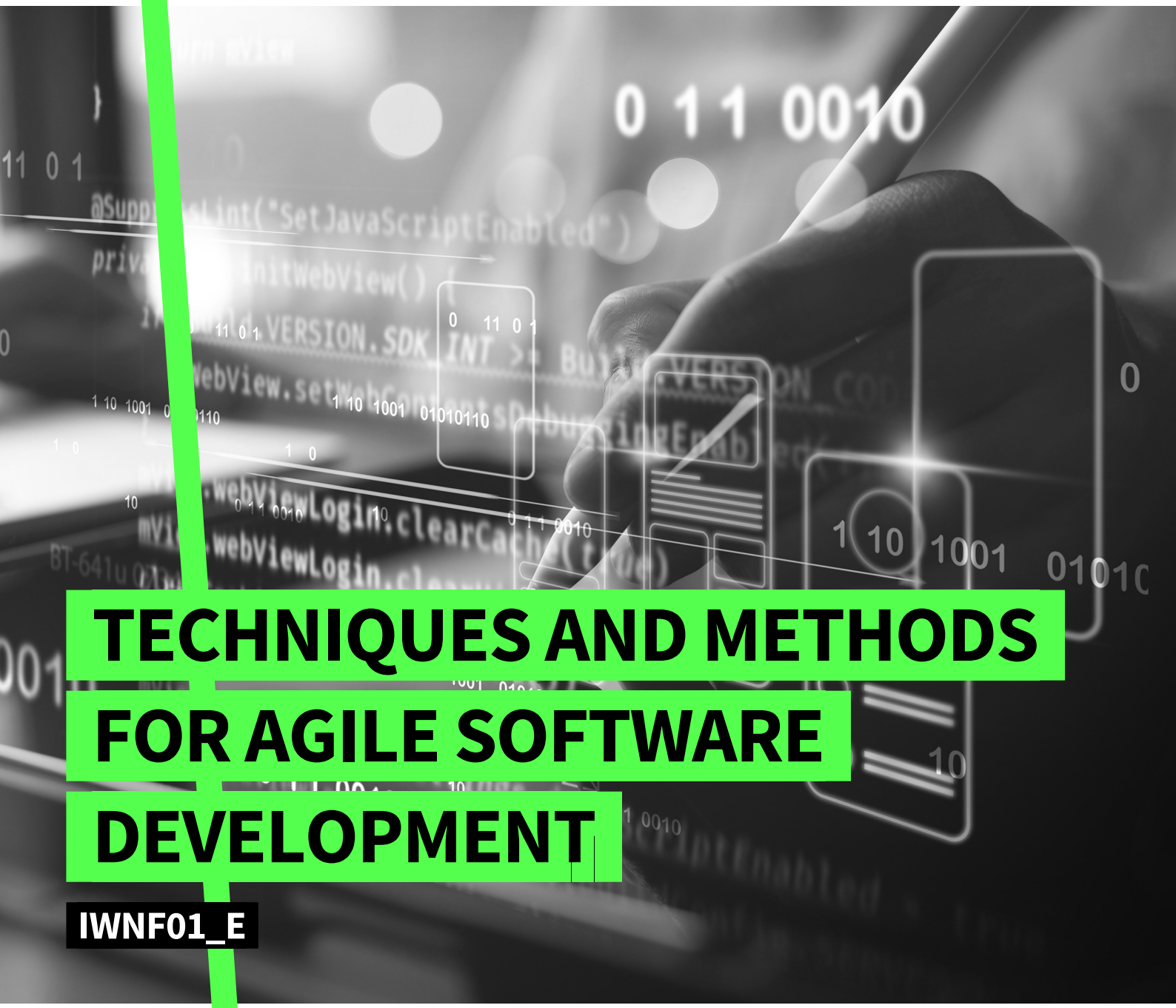
Course Book



TECHNIQUES AND METHODS FOR AGILE SOFTWARE DEVELOPMENT

IWNF01_E

iu
INTERNATIONAL
UNIVERSITY OF
APPLIED SCIENCES

TECHNIQUES AND METHODS FOR AGILE SOFTWARE DEVELOPMENT

MASTHEAD

Publisher:
IU Internationale Hochschule GmbH
IU International University of Applied Sciences
Juri-Gagarin-Ring 152
D-99084 Erfurt

Mailing address:
Albert-Proeller-Straße 15-19
D-86675 Buchdorf
media@iu.org
www.iu.de

IWNF01_E
Version No.: 001-2023-1013
Translated by Sandra Rebholz

TABLE OF CONTENTS

TECHNIQUES AND METHODS FOR AGILE SOFTWARE DEVELOPMENT

INTRODUCTION

WELCOME

SIGNPOSTS THROUGHOUT THE COURSE BOOK

This course book contains the core content for this course. Additional learning materials can be found on the learning platform, but this course book should form the basis for your learning.

The content of this course book is divided into units, which are divided further into sections. Each section contains only one new key concept to allow you to quickly and efficiently add new learning material to your existing knowledge.

At the end of each section of the digital course book, you will find self-check questions. These questions are designed to help you check whether you have understood the concepts in each section.

For all modules with a final exam, you must complete the knowledge tests on the learning platform. You will pass the knowledge test for each unit when you answer at least 80% of the questions correctly.

When you have passed the knowledge tests for all the units, the course is considered finished and you will be able to register for the final assessment. Please ensure that you complete the evaluation prior to registering for the assessment.

Good luck!

BASIC READING

Rubin, K. S. (2012). *Essential scrum: A practical guide to the most popular agile processes*. Addison-Wesley Professional. http://search.ebscohost.com.pxz.iubh.de:8080/login.aspx?direct=true&db=cat05114a&AN=ihb.52492&lang=de&site=eds-live&scope=site

Sutherland, J. & Coplien, J. O. (2019). *A Scrum book: The spirit of the game.* The Pragmatic Programmers. http://search.ebscohost.com.pxz.iubh.de:8080/login.aspx?direct=true&db=cat05114a&AN=ihb.52079&site=eds-live&scope=site

FURTHER READING

UNIT 1

Armour, P. G. (2000). The five orders of ignorance. *Communications of the ACM, 43(*10*),* 17–20. http://search.ebscohost.com.pxz.iubh.de:8080/login.aspx?direct=true&db=bsu&AN=11950405&site=eds-live&scope=site

UNIT 2

Schwaber, K. & Sutherland, J. (2020, November). The definitive guide to Scrum: The rules of the game*. Scrum Guides.* Available online

UNIT 3

Rubin, K. S. (2012). *Essential scrum: A practical guide to the most popular agile processes*. Addison-Wesley Professional. Chapter 15 http://search.ebscohost.com.pxz.iubh.de:8080/login.aspx?direct=true&db=cat05114a&AN=ihb.52492&lang=de&site=eds-live&scope=site

UNIT 4

Fowler, M. (2019, May 21). Technical debt. *Thoughtworks*. Available online

UNIT 5

Baumgartner, M., Klong, M., Mastnak, C., Pichler, H., Seidl, R. & Tanczos, S. (2021). *Agile testing: The agile way to quality*. Springer. Chapters 5 & 6 http://search.ebscohost.com.pxz.iubh.de:8080/login.aspx?direct=true&db=edsebk&AN=3032464&site=eds-live&scope=site

UNIT 6

Kim, G., Humble, J., Debois, P. & Willis, J. (2016). *The DevOps handbook: How to create world-class agility, reliability, and security in technology organizations* (pp. 43–51). IT Revolution Press. http://search.ebscohost.com.pxz.iubh.de:8080/login.aspx?direct=true&db=cat05114a&AN=ihb.48152&site=eds-live&scope=site

LEARNING OBJECTIVES

In **Techniques and Methods for Agile Software Development** you will be introduced to the concept of agility and learn how it can be applied to the software development process. We will begin with an introduction to and discussion of the basic characteristics and principles of this approach, and you will learn to distinguish between classic and agile approaches to software development.

Building on this basic knowledge, we will then move on to a practical overview, exploring how agile software engineering can be conducted in small projects and teams. Approaches to applying agile principles in larger projects will also be presented. Finally, you will learn techniques for conducting core activities in software engineering in an agile way. Here, the focus will be on the domains of testing, delivery, and deployment.

UNIT 1

CHARACTERISTICS AND PRINCIPLES OF AGILITY

STUDY GOALS

On completion of this unit, you will be able to ...

- recognize the unique features and challenges of software projects.
- identify and manage the sources of risk and uncertainty in the context of software projects.
- understand the difference between agile and classic software development.
- define the principles of agility in the context of software development.

1. CHARACTERISTICS AND PRINCIPLES OF AGILITY

Introduction

Software engineering
This is the application of engineering approaches to the development of software.

This unit introduces and discusses the term "agility" in the context of **software engineering**. After presenting some typical challenges of software projects, uncertainty and ignorance are analyzed in detail. The next section focuses on outlining and comparing agile and traditional software engineering. Finally, agile principles (as they are commonly accepted in the field of software development) are presented and explained in detail.

1.1 Features and Challenges of Software Projects

Software projects share some traits with traditional industrial and manufacturing projects, but they also have unique features that make them challenging to plan and execute successfully when using traditional project management principles.

Characteristics of Software Systems

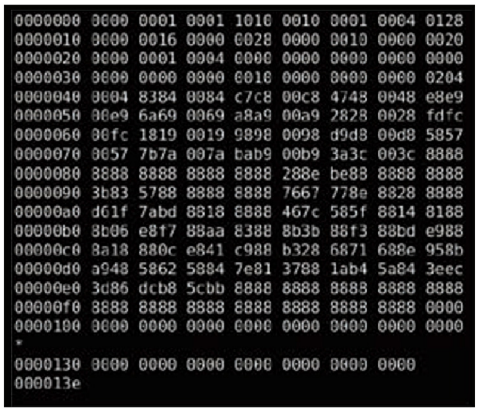
Software system
A software system consists of computer programs, configuration data, documentation, and support websites.

A common characteristic of all **software systems** is their complexity. An enterprise software system

- includes many technical components and subsystems.
- is connected to many other systems via technical interfaces.
- supports a wide range of functions.

Moreover, software systems are intangible. A completed software system manifests itself as a long series of 0s and 1s on a storage medium. This is a challenge in many ways: defects cannot be seen, and you can never look directly at the components and their dependencies during the planning, construction, and use of the software. Neither the project manager nor the customer can get an idea of the actual “construction progress” of the software by inspecting a construction site (Sommerville, 2016, p. 18). Because it is difficult to represent software systems in “real-world” terms, software engineers must be able to develop abstract, logical structures.

Figure 1: Software System versus Machine



Source: Created on behalf of the IU (2023), based on Jaguar MENA (2014), CC BY 2.0; Metoews (2016), CC BY-SA 3.0.

When describing the characteristics of specific software systems, the focus is on their respective functional and non-functional characteristics. Functional characteristics describe the set of functions that the system offers to its users. Non-functional characteristics are also referred to as the quality attributes of software systems. They include the system's performance efficiency, reliability, usability, security, compatibility, portability, and maintainability (International Organization for Standardization [ISO], 2011).

Characteristics of Software Projects

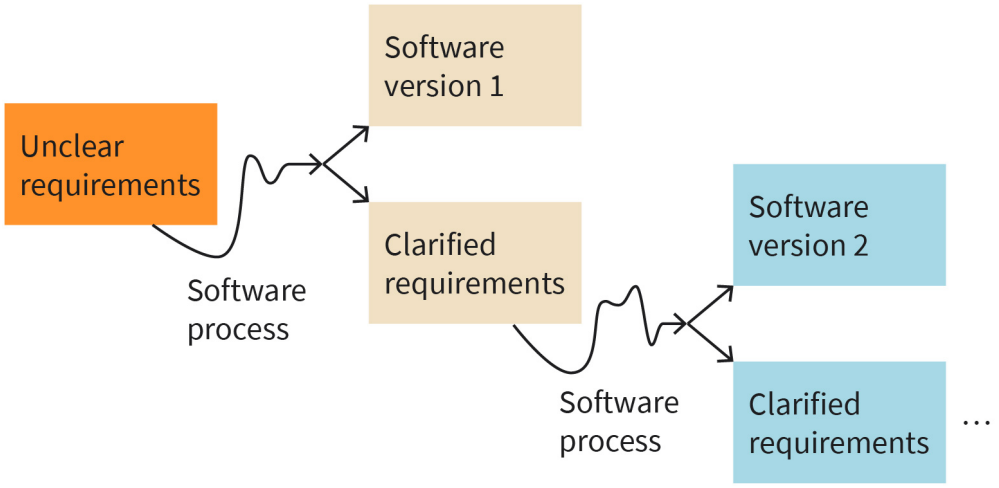
In industrial processes that mostly involve the production of material goods, many work steps can be planned precisely in terms of duration and resource requirements. In contrast, precise planning is rarely possible in software engineering projects. Software engineering involves developing unique products, with features specific to the software. This process does not involve the manufacturing of a previously specified product according to a proven sequence of working steps. Instead, the goal is to envision and develop a new product that meets the needs of the customers. Predicting the exact costs, specific scope of functions, and technical design of a software system upfront is not possible; these factors can only be precisely determined at the end of a project.

One main characteristic of software development is that it is a knowledge-driven process. We will illustrate this using an example. Complex software projects are often started with an overarching business goal, for example the introduction of a self-care portal for insurance company customers. However, which specific functions the system has to fulfill in detail, which screen dialogs are required, and how other software systems can be integrated cannot be precisely specified at the beginning of the project. The parties involved will only gain a full understanding of these aspects as the project progresses.

A software engineer is faced with the following dilemma: If you start to develop a system for which the requirements are not clear, you run a high risk that the application created does not implement the desired functions. However, at the beginning of the development

process, users are often unclear about which functions are needed. Usually, their needs will become clearer once they have tested an initial version of the system. The figure below shows the interaction between recognized requirements and the software process.

Figure 2: Software Engineering is Strongly Knowledge-Driven



Source: Created on behalf of the IU (2021).

The knowledge-driven nature of software engineering can present a dilemma that is further complicated by other factors: for example, the fact that there are usually many stakeholders, each with their own requirements for the system, often spread over several organizational units. This means that many additional requirements may be identified during the software process.

In addition to the stakeholders, new legal (compliance) requirements, technological developments, and the market situation can also significantly influence the requirements for software systems. This leads directly to a typical reason for failed software projects: changing and conflicting requirements.

The development team must understand the requirements formulated by the stakeholders from a business perspective so that it can deliver a usable software system. Because today's enterprise software systems are usually integrated into complex enterprise architectures via many interfaces, the development team must also be able to recognize the conceptual relationships across system boundaries. In addition, a project team with domain knowledge can cope more reliably with unclear requirements, as it can independently identify gaps or errors in the requirements and offer the stakeholders targeted solutions.

Risks in Software Engineering

Due to the complexity and immateriality of enterprise software systems, as well as the knowledge-driven nature of software projects, there are a number of typical risks associated with software projects. These risks can be categorized into risks that lead to the ter-

mination of the project, risks that are related to the applicability of the software, and risks that are related to software maintenance. These types of risks are explained in more detail in the following paragraphs.

Project termination risks

There are many risks that could lead to the termination of the project. The following are examples of project termination risks:

- Requirements turn out to be unrealizable in the course of the project. Typical causes are constantly changing key requirements; excessive expectations of the system; or technical, organizational, and compliance conditions that were recognized too late.
- The costs for the project go over budget before the system is ready for use and the project team cannot provide any reliable information about future costs.
- Members or different organizational units within the project have fundamentally different opinions or are at odds. They no longer trust each other and constructive cooperation is no longer possible.

Software applicability risks

Risks that affect the functionality of the software or the quality attributes, or that relate to a software product not fulfilling the needs of the target users are known as "software applicability risks". The following situations could present such risks:

- During deployment of the software, it turns out that important business functions are missing or have been implemented incorrectly.
- Too much focus is placed on using the latest technologies instead of addressing the actual needs of the user (technology-centricity).
- Quality requirements stated by the customer are not met. The system reacts too slowly when there are high numbers of users, the transmission of critical data is not adequately secured, or the timeframe for nightly jobs (processing large amounts of data) cannot be adhered to.
- The system is not accepted by users because, for example, data input takes too long, operation is too cumbersome and complicated, or the user interface has been redesigned so that familiar elements are difficult or impossible to find.

Maintenance risks

Professional software is in use for many years and must be maintained and developed to adapt to changing business requirements. In order to carry out maintenance measures economically, the software has to be written and documented in a way that makes modifications and upkeep as simple as possible. Examples of risks related to maintenance include the following:

- The team responsible for maintenance cannot envision the consequences of its actions. Dependencies and relationships within the system cannot be recognized based on the documentation.

- The internal structure of the system has degenerated over the years due to ongoing small maintenance work and adjustments. Even simple maintenance work requires enormous effort that is disproportional to the economic benefit.
- There is no knowledge available about the technologies used in the legacy system. The original developers of this system have retired and there are no specialists who have experience with the old programming languages and operating systems.

Challenges in Software Engineering

Several challenges can arise in the context of software development projects for enterprise information systems. In this section, we will discuss the following challenges in software engineering:

- estimation of effort
- technological uncertainty
- communication among stakeholders
- conflicting goals
- handling complexity

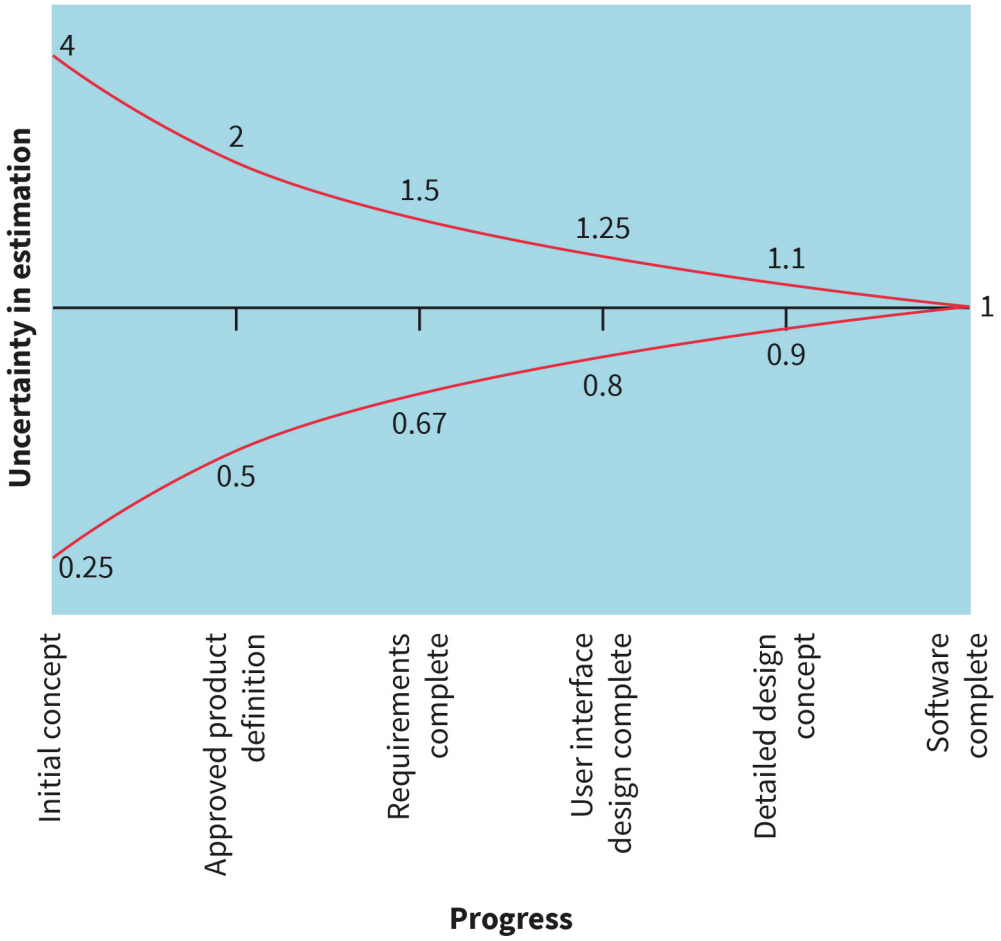
Estimation of effort

Cone of uncertainty
The cone of uncertainty represents the uncertainty associated with effort estimates in software projects.

A central challenge in enterprise software projects is dealing with and considering uncertainty throughout the course of the project. To illustrate uncertainty with respect to effort estimation in software projects (economic uncertainty), the figure below shows the **"cone of uncertainty"**, determined using statistical methods (Bauman, 1958). The vertical axis (y-axis) represents the degree of uncertainty, while the horizontal axis (x-axis) indicates the course of a software project over time. The project starts on the left and is finished on the right.

The graph shown in the figure below has the shape of a cone rotated by 90 degrees. The earlier the total costs are estimated in a project, the higher the deviation from the actual effort at the end of the project. As the project progresses, an estimate of the total costs becomes more precise, but it is only at the end of the project that it is possible to determine the level of effort exactly. In contrast to production processes, in which the consumption of resources and the duration of the individual work steps can be calculated, a precise forecast is not usually possible in software projects.

Figure 3: Cone of Uncertainty



Source: Tobias Brückmann (2021), based on Bauman (1958).

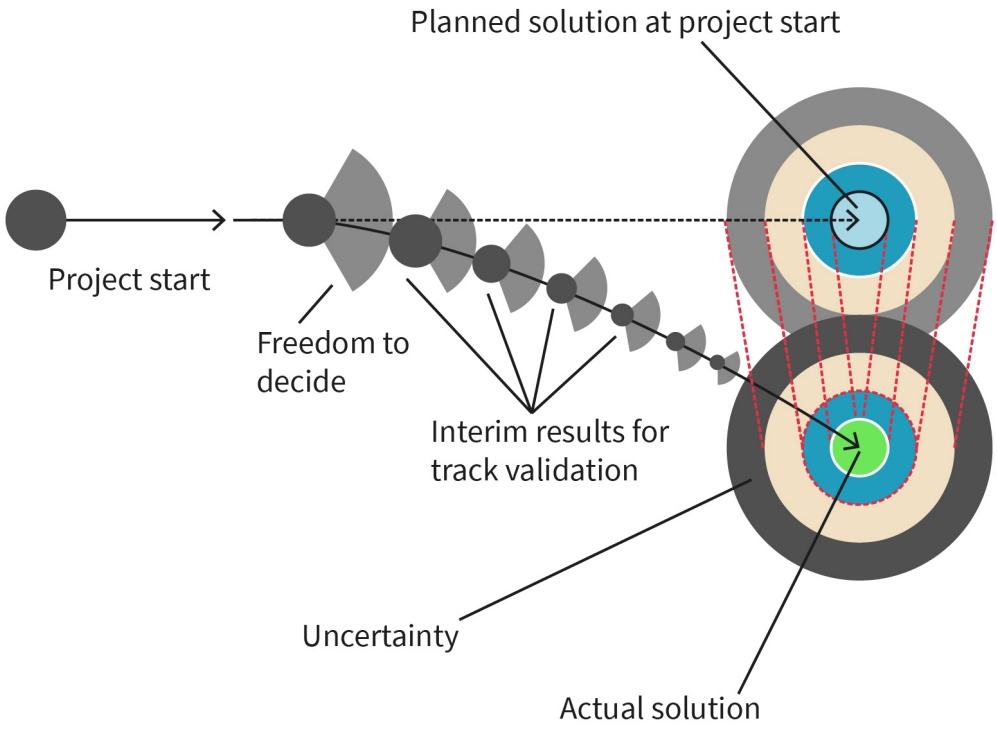
Technological uncertainty

A technological vision can be created at the beginning of a software project, but the actual technical solution can only be precisely determined at the end. The figure below illustrates the course of technical uncertainty in a software project. At the beginning of a project, there is still a lot of freedom to decide which technologies to use.

Technical decisions are made by the software architect based on the current state of knowledge, to match the known requirements as closely as possible. However, since requirements are only identified in the course of the project, the interim results must be checked continuously. The technical solution can then be further adapted and expanded within the scope of all constraints.

As a rule, the possible scope for decision-making with regard to technical solutions becomes smaller as the project progresses, until a concrete solution is worked out at the end. This also means that technical uncertainty (like economic uncertainty) must be endured by the project team.

Figure 4: Technological Uncertainty



Source: Created on behalf of the IU (2021).

Communication among stakeholders

The inclusion of various stakeholders (each with different backgrounds, knowledge, and interests) represents a further important challenge for enterprise software projects. Care must be taken to establish and maintain willingness to cooperate. New findings and if necessary, changes to the plan, must be communicated to all relevant stakeholders in a manner that they can understand. In addition, the relevant stakeholders must be actively involved in making decisions and in assuring the quality of artifacts generated in the software process.

Conflicting goals

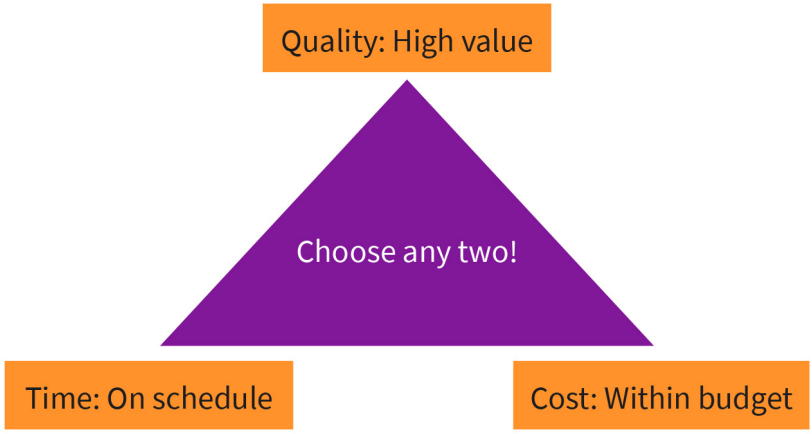
Magic triangle
The magic triangle illustrates competing goals.

Customer satisfaction is the top priority of software projects, but decisions made within software projects also depend on quality, time, and cost factors. The figure below illustrates a typical decision dilemma with the help of the **magic triangle**: If you improve one target parameter, it is usually at the cost of the other two target parameters.

For example, if quality is very important to a project, quality assurance activities may come at the expense of timely completion and budget. If the focus is on project costs and all that deadlines are met, then fewer resources remain for quality assurance. However, if a high-quality project is to be delivered on time, then the budget may not be adhered to.

Due to the uncertainty presented, together with the ongoing gain in knowledge during the project, the size of the "magic triangle" also changes continuously in software projects. This results in frequent coordination and decision-making processes between the project team and the stakeholders.

Figure 5: The Magic Triangle: Quality, Time, and Cost



Source: Created on behalf of the IU (2021).

Handling complexity

The physical complexity of enterprise software systems (many technical components, many interfaces to other systems, and many functions to be implemented), in combination with the immateriality of software, places high demands on human abstraction. Because the internal structures of software are not visible, the relationships between the various software components are illustrated with the help of graphical models (for example, using standardized notations like the Unified Modeling Language [UML]). Inferring the structure and behavior of the software from a graphic model requires intense intellectual effort.

The use of **software models** is a key practice in software engineering to which there are currently no sensible alternatives. Various types of models that illustrate static and dynamic aspects of software systems in varying degrees of detail have been developed in order to master complexity. Models of both the organizational planning (resources and activities) and the technical planning (specification and graphic software models) should only be created to the extent that is appropriate for the current phase of the project.

Software model: This is a graphical representation of the design of a software system.

In software projects, attempts are often made to compensate for uncertainties by creating a plan that is as precise as possible, or a model that is as detailed as possible and incorporates all potential eventualities. In some cases, an overly detailed project plan is drawn up,

with the aim of predicting which activities are to be carried out for the entire project, including details of the people and effort involved. In others, the business requirements and solutions of the software system are analyzed on a detailed level and documented in the form of extensive models. This phenomenon is called **"analysis paralysis"**. Complexity paired with uncertainty harbors the risk of inappropriately expensive analysis activities. An attempt is made to hide the uncertainty by producing complex analysis results, for example, in the form of large and complex software models. Often, the team does not simply try something, but first analyzes everything in detail. The danger of analysis paralysis is that many resources are tied up, which are then missing when implementing an initial software version. In addition, the elaborate analysis models that were created at the beginning often become obsolete due to knowledge gained during the implementation and presentation of the first software version.

Analysis paralysis
This is a phenomenon of inappropriately complex analysis activities.

1.2 Classification of Uncertainty

In enterprise software engineering it is essential to make decisions in the face of uncertainty and lack of knowledge. This section introduces different types of uncertainties that are relevant in the domain of software engineering. It also provides an overview of "The Five Orders of Ignorance," a model that classifies ignorance in terms of different layers.

Types of Uncertainty

There are different types of uncertainty that are inherent in complex product development (Rubin, 2013, p. 70):

- **End uncertainty ("what?" uncertainty)** is uncertainty about the properties of the final software system, such as the functions implemented, the technical architecture, or the final quality of the system. This uncertainty is present until development is finished.
- **Mean uncertainty ("how?" uncertainty)** is uncertainty about methods, tools, processes, and activities that are applied in software development. This category comprises all uncertainties related to the software construction process including cost estimation. Similar to end uncertainty, it is only after the software development project has finished that the exact development process is known. The concrete activities, the sequence in which they were executed, and the time required is only known after the project has ended.
- **Stakeholder uncertainty ("who?" uncertainty)** is uncertainty about the stakeholders involved in the development process and the future end users of the product. At the beginning of a project, the interests of the various stakeholders and how they influence project-related decisions are often unclear. It is only at the end of the project that stakeholder uncertainty can be resolved. This is particularly true for startups, where the uncertainty related to the future clients (who will use and pay for the system) plays a major role.
- **Domain uncertainty ("wherein?" uncertainty)** is uncertainty about the domain, the subject area, or the business area in which the system will later be used. This category is comprised of the uncertainty related to typical business domain objects, business processes, and business rules.

In light of the fact that many uncertainties can only be resolved at the end of a software project, it is important for software development activities to be designed in a way that addresses these uncertainties during the execution of the project.

Levels of Ignorance

Ignorance is another aspect that must be taken into consideration when analyzing and assessing uncertainty. Armour (2000) proposed a model called "The Five Orders of Ignorance" that classifies ignorance into different layers. The model is based on the following assumptions (Armour, 2000, p. 7):

- Software is a medium for storing knowledge.
- The product of software engineering is knowledge that manifests itself in a software system.
- Software engineering activities are activities for acquiring knowledge (see also "Types of Uncertainty") and for translating this acquired knowledge into program code.

Software engineering is considered an ongoing process of acquiring knowledge and reducing ignorance. In this context, Armour (2000) describes five different levels of ignorance. These levels do not refer to knowledge in general, but rather, to knowledge or ignorance in a specific context, for example, in a concrete scenario, with regard to specific domain knowledge, or regarding a special type of uncertainty.

0th Level of ignorance: lack of ignorance

On this level, knowledge is available. There is no ignorance at all. Everything that needs to be known to write the program code is known and a developer can work without acquiring new knowledge.

1st Level of ignorance: lack of knowledge

On this level, one is aware that certain knowledge is not available and has to be acquired. The developer is able to ask a specific question in order to acquire the missing knowledge they need in order to write the program code.

2nd Level of ignorance: lack of awareness

On this level, one is aware that some knowledge is missing, but it is not known what exact piece of knowledge is missing. In this case, it is not possible to ask specific questions in order to acquire the missing knowledge. For example, a developer knows that there is a subject area that needs to be explored. However, they do not know any details about this subject area, and thus, are not able to ask any specific questions about it.

3rd Level of ignorance: lack of process

On this level, one is aware that some knowledge is missing. However, it is not known how to acquire the missing knowledge in a structured and timely way: A developer may be familiar with the phases of the software life cycle . However, they developer may not know

the concrete activities that must be executed within the individual phases. Consequently, they are missing the procedure to acquire the necessary knowledge in the required timeframe.

4th Level of ignorance: meta ignorance

On this level, it is known that there are software engineering activities that deal with acquiring knowledge. However, it is not known that there are different levels of ignorance. When trying to acquire knowledge, any known technique for doing so is applied on all levels of ignorance in the same way.

In the following table, the levels of ignorance are explained in relation to the different types of uncertainties introduced above. For every level of ignorance, an example is provided in the contexts of mean uncertainty, end uncertainty, and stakeholder uncertainty.

Table 1: Examples of the Levels of Ignorance

	Context: mean uncertainty	**Context: end uncertainty**	**Context: stakeholder uncertainty**
0th Order ignorance (0-OI) Lack of ignorance. "I know something specific. I know the answer."	I can apply the suitable elicitation technique in **requirements engineering (RE)**.	I know how to implement the algorithm for calculating premiums. I know all required data and all business rules.	I know all relevant stakeholders, their interests in the project, and the influence they have on the project.
1st Order ignorance (1-OI) Lack of knowledge. "I am missing a specific piece of knowledge. I have a specific question."	I know that there are different techniques for requirements elicitation and how to learn them. **Achieving 0-OI:** Learn the techniques.	I know that I must implement an algorithm for calculating premiums, and I know what kind of information I must research. **Achieving 0-OI:** Obtain the missing information.	I know all the stakeholders of a project. I know how to research their interests and their influence on the project. **Achieving 0-OI:** Obtain information on the stakeholders.

Requirements engineering: This comprises all activities for the determination, documentation, negotiation, and administration of requirements.

	Context: mean uncertainty	Context: end uncertainty	Context: stakeholder uncertainty
2nd Order ignorance (2-OI) Lack of awareness. "I am not aware of the piece of knowledge that I am missing. I cannot ask a specific question."	I know that RE is a core activity of software engineering. However, I do not know the concrete activities and techniques that are applied in RE. Because of that I do not know that there are different techniques for requirements elicitation. **Achieving 1-OI:** Learn the concrete activities and techniques of RE.	I know that I need to implement an IT system for calculating premiums for an insurance company. However, I do not have any experience in the implementation of such systems. Because of that, I do not know that I must implement an algorithm for calculating premiums. **Achieving 1-OI:** Learn about the business architecture, create or learn about the system architecture.	I have compiled a list of relevant stakeholders. However, I do not know that some stakeholders are missing from the list. **Achieving 1-OI:** Learn methods and techniques for stakeholder management.
3rd Order ignorance (3-OI) Lack of process. "I do not know how to acquire missing knowledge in a timely manner. I do not know how to find out what to ask."	I know that RE is a core activity of software engineering. However, I do not know how to acquire competencies in the field of RE in a structured way. **Achieving 2-OI:** Learn how to systematically acquire competencies in the field of RE.	I know that I need to implement an IT system for an insurance company. However, I do not know what type of system I should implement. **Achieving 2-OI:** Acquire knowledge on the type of system to be implemented.	I am not aware that relevant stakeholders have a strong influence on project success. **Achieving 2-OI:** Learn how to systematically acquire knowledge on stakeholder management.
4th Order ignorance (4-OI) Meta ignorance. "I do not know that there are different levels of ignorance."	I know that software engineering is involves activities for acquiring knowledge. **Achieving 3-OI:** Learn how the field of software engineering is structured.	I know that the result of the software engineering process is program code. **Achieving 3-OI:** Acquire knowledge on the context in which the software system shall be developed.	I know that people are relevant for software development processes. **Achieving 3-OI:** Acquire knowledge of how the domain "the role of people in software projects" is structured.

Source: Created on behalf of the IU (2023).

It is critical to know the level of ignorance of a project team related to the type of uncertainty in a certain project context. Having this knowledge allows you to adequately manage the activities in a software project. Typically, the following rule applies: the higher the level of ignorance, the higher the project risk. For example, if you create highly detailed project plans or system designs with a high level of ignorance, these artifacts are created based on assumptions and not on concrete knowledge. Agile software engineering addresses exactly this problem.

1.3 Comparison of Agile and Classic Software Development

Typically, the terms “agility” or “agile software development” do not denote concrete techniques or processes; instead, they refer to a mindset or to design principles for software engineering activities. There is no single, agreed upon definition of the term agility. Often, agility is seen as a value or attitude that serves as the basis for designing daily project activities. In contrast with agile software engineering, the traditional or classic approach to software engineering is often referred to as "plan-driven". The table below provides a comparison of both approaches and their characteristics.

Table 2: Comparing Agile and Plan-Driven Approaches

	Agile software development	**Classic software development**
Typical terms	Agile, agility	Plan-driven
Implementing software process model	Scrum	Waterfall model, V-Model
Management principle	Knowledge-based: Decisions are made based on knowledge (for example, through obtaining feedback).	Assumption-based: Decisions are made based on assumptions.
Characteristics	• Short release cycles • Short project phases in high frequency • Specification details only for selected functions • Early first release • Low level of detail in planning and management artifacts • Adaptation of management and product artifacts is frequent, expected, and planned • Changes to the plan are recieved positively	• Long release cycles • Relatively long project phases in low frequency • Specification shows a high level of detail at the beginning of the implementation phase • First release is quite late in the project • High level of detail in planning and management artifacts • Adaptation of management and product artifacts is exceptional and unplanned • Changes to the plan are received negatively
Feedback loop	Collects and considers continuous feedback based on interim project results	Obtains feedback only at the end of the project based on the final project result

Source: Created on behalf of the IU (2023).

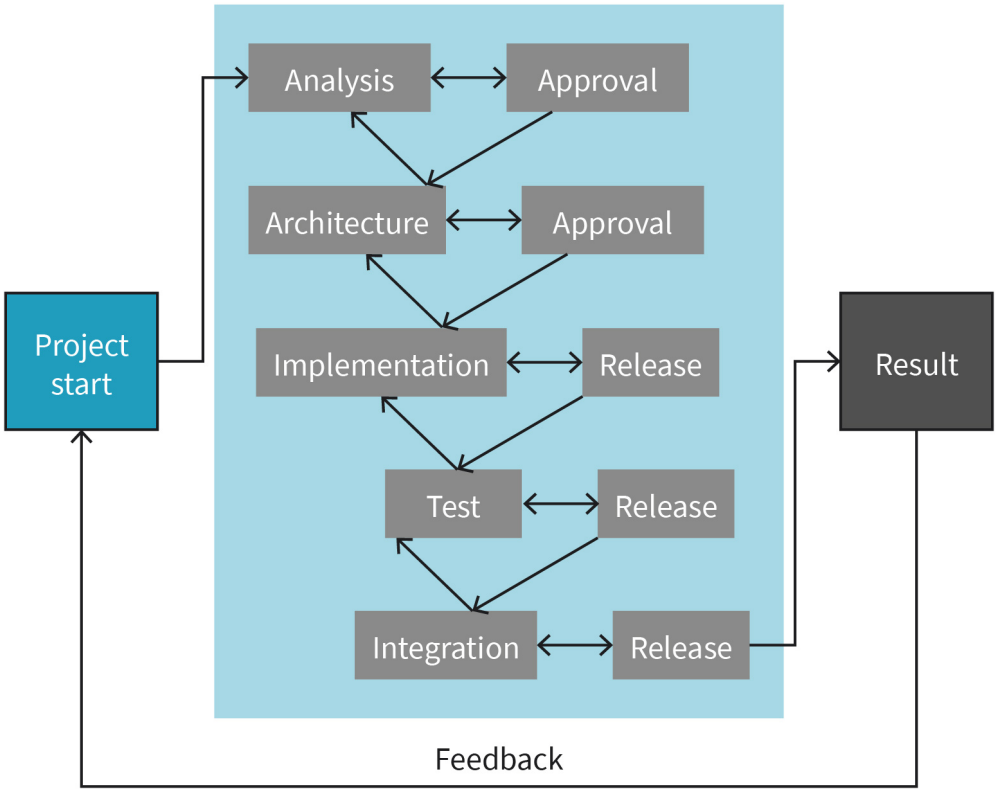
The fundamental difference between the approaches is the basis on which important decisions are made. Agile software development aims at making important decisions based on knowledge gained while the project is running. So, this approach is **knowledge-driven.** Conversly, classic software development relies on plans that were created in early phases of the project or even before the project started. In this approach, assumptions about the progression of the project and the desired results are made based on previous

Knowledge-driven
In this approac, decisions are made based on actual knowledge.

experience and on the current state of knowledge of the people involved. These projects are known as **asumption-driven** because they are conducted based on assumptions about the future.

asumption-driven
Projects in which decisions are made based on assumptions about the future are called assumption-driven.

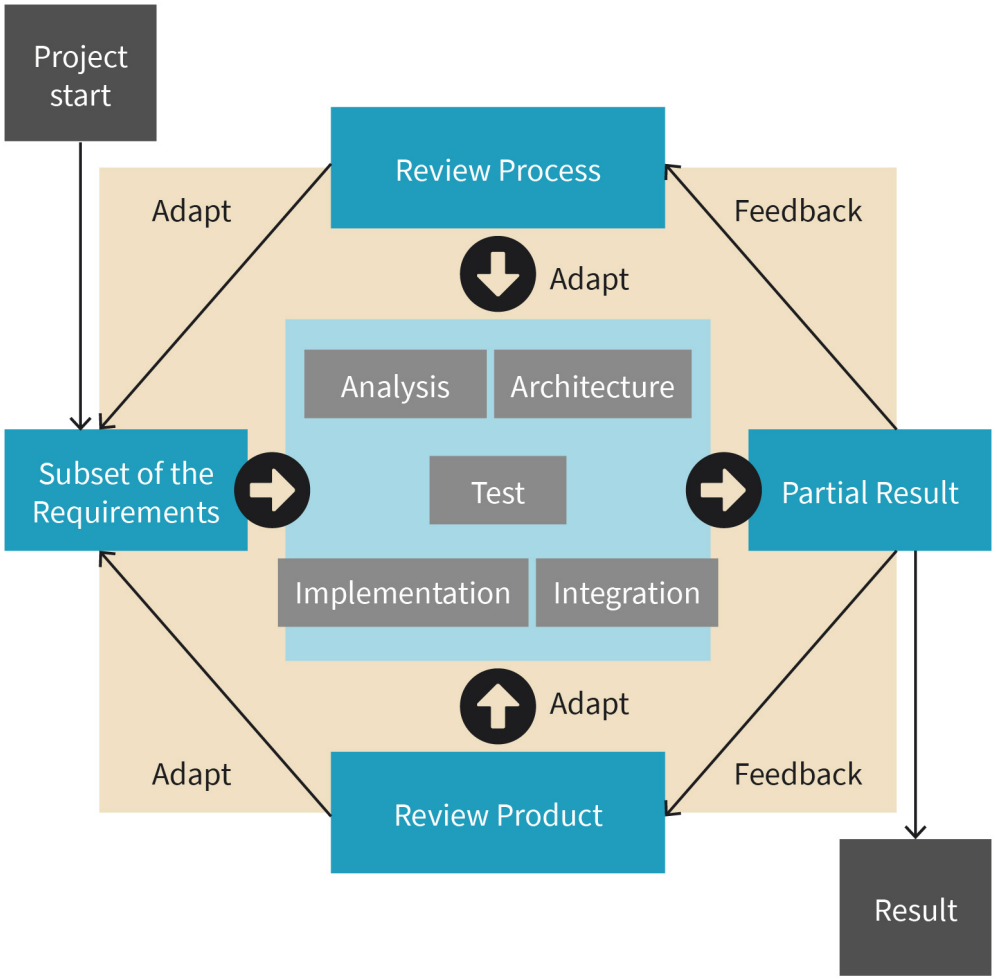
Figure 6: Assumption-Driven Software Engineering



Source: Created on behalf of the IU (2023).

The figure above illustrates an example of knowledge-driven software development. Initial requirements are analyzed, and only a subset of those requirements is selected for the first cycle of development. Based on the interim project result, both the interim result and the organization of the development activities are reviewed and reflected. If necessary, the set of requirements for the next cycle is modified based on the newly gained insights. The feedback loop between result and requirements is an essential element of the development process: It enables control of the process based on the knowledge gained during project execution.

Figure 7: Knowledge-Driven Software Development



Source: Created on behalf of the IU (2023) based on Rubin (2013, p. 36).

There is no one-size-fits-all approach when comparing both approaches to software development. The constraints of a project and the type and quality of uncertainty and ignorance at the start of the project should be considered when determining the best approach to designing the concrete software engineering activities. A small, simple project involving an experienced team adapting a system that they have built themselves can mostly be managed based on assumptions. Conversly, a project that aims at building a system for a new business domain based on a new technology platform and that is implemented by a newly formed team, will involve exposure to a high level of uncertainty of different types. Such projects should mostly be managed based on the knowledge that is gained during project execution.

Agility as Design Principle

Agility can be understood as a design principle for software processes. There is no single agile software process framework that can be used as a basis for all software projects. However, there are a number of guiding principles that can be applied when defining concrete activities, roles, and decision-making pathways for agile software processes. What's more, medium-size and large-scale enterprise software projects are dependent on other projects and management decisions, which are not necessarily organized according to agile principles. More recent approaches aim at aligning the business processes of the whole enterprise with agile principles. These approaches strive for **business agility**, and define the principles, practices, and competencies required to realize agility on the level of the enterprise and in large-scale projects. Examples are the Disciplined Agile® toolkit (Project Management Institute, Inc., 2022) and the Scaled Agile Framework® (Scaled Agile, Inc., 2021d). Because agility is not the "correct" approach per se, but one of several ways for organizing software projects, the challenge of finding the right balance between knowledge-driven management and assumption-driven management of software projects remains. Finding the right balance depends on the project size, the given constraints, and the competencies of the team members.

Business Agility
The ability of an enterprise to respond quickly to market changes by applying agile principles and practices.

1.4 Principles of Agility

How it all Started: The Agile Manifesto

In 2001, a group of innovative software engineers published the "Manifesto for Agile Software Development" (Beck et al., 2001). All the authors of the manifesto were experienced, well-known software engineers. From the moment of its publication, the manifesto was seen as an answer to complex software process frameworks such as the Rational Unified Process (RUP) or the V-model that were widely used at the time. A key thesis underlying the manifesto states that by focusing on the adherence to organizational rules and specifications, these software process frameworks prevent appropriate reactions to changes within the project, and thereby hinder its successful execution.

The Values of Agile Software Development

The agile manifesto is comprised of four agile values, formulated by the authors based on their professional experience in software engineering projects. The manifesto can be summed up as follows:

Manifesto for Agile Software Development

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> - Individuals and interactions over processes and tools
> - Working software over comprehensive documentation
> - Customer collaboration over contract negotiation

- Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more. (Beck et al., 2001)

Individuals and interactions over processes and tools

Although applying processes and using tools is important, it is not more important than direct collaboration and communication between stakeholders in a project. As already stated, the success of a project is highly dependent on the collaboration between the people involved. A well-defined process model and tools are beneficial, but the use of tools can only become truly effective if all the people involved collaborate well.

Working software over comprehensive documentation

The main expectation of customers is that a software development project will result in a system that fulfills their requirements. It is only helpful to document the activities in the software development process well and provide a comprehensive documentation of the system functionalities if the system actually works. Early software process paradigms like the waterfall model or the V-Modell impose strict rules regarding documentation within the development process. This ties up many resources that are subsequently not available for other development activities.

Customer collaboration over contract negotiation

IT projects are usually major investments for an enterprise. If a software development project involves a client and an external service provider that creates the new software system for the client, the project must be governed by a legally binding contract. Fulfilling the exact requirements, functions, dates, and costs that have been specified in the contract rarely leads to a result that addresses what the client really needs. This is especially the case in large-scale IT projects that are often characterized by domain-specific and technical uncertainties. Therefore, the clients must be actively involved in the planning and in the functional design of the software system during the course of the project.

Responding to change over following a plan

Newly identified requirements and changes in the organizational or technological context during development demand a high degree of adaptability and the willingness to accept change. Changing system requirements and changes in the project plan, are a reality in every project. Often, the priority of requirements is changed, and previously low-priority requirements are now needed urgently and must be implemented as soon as possible. This requires that the project team is both willing to and able to react to the changes in the project plan appropriately.

In defining the principles of agile software development (Beck et al., 2001), a kind of organizational strategy is outlined, which serves as a basis for designing concrete, individual software processes. Although you cannot derive a process from these principles directly, they represent a guideline for designing activities in software engineering as well as a checklist for reviewing these activities.

Rubin (2013) has compiled a list of agile principles that are derived from the agile manifesto and other relevant resources. Rubin (2013) groups the principles according to the following six categories:

1. Variability and uncertainty
2. Prediction and adaptability
3. Validated learning
4. Work in progress (WIP)
5. Progress
6. Performance

Variability and Uncertainty

This category summarizes principles that explicitly address the uncertainties in a software process.

Variability of the development process

As opposed to industrial production processes, a software process is not about creating several instances of the same product, but about creating a unique product. All software systems differ from each other, even though their purpose might be the same. Therefore, the development process should be variable in a way that allows it to adapt to the software project at hand. This also holds for processes that have already been launched.

Combine iterative and incremental development

In **iterative development**, the software development process is structured in cycles. Every cycle (or iteration) of product development results in an interim product that is reworked and improved in the next cycle based on the knowledge gained during the ongoing development. By doing so, the product is refined and optimized iteratively. In **incremental development**, the product is broken down into parts. In every development cycle, a working version of the product is developed that provides part of the required functionality. This working version of the software is called an increment (Agile Alliance, 2022.). Combining iterative and incremental development approaches leads to a software process that enables the team to react appropriately to insights that are only gained during the development process.

Iterative development
This is an approach to software development that is organized into cycles aiming at optimizing and refining the work product with every cycle.

Incremental development
This is an approach to software development that successively delivers working versions of the product enhanced with added functionality in every version.

Continuous adaptation through inspection and transparency

Inspection and reflection activities are conducted during the development process. This affects both the software system and the software process. The goal of these activities is ensuring a high product quality and a continuous improvement of the collaboration

among stakeholders. An important prerequisite for this is transparency, in the sense that all information needed to develop the software system is available to all stakeholders (Schwaber & Sutherland, 2020).

Address all types of uncertainty at the same time

Several types of uncertainty are dealt with simultaneously instead of sequentially. This means that you would not eliminate mean uncertainty first and then eliminate product uncertainty. Instead, you would reduce every type of uncertainty gradually by applying incremental development steps.

Prediction and Adaptability

This category summarizes principles that focus on the opposites of prediction and planning on the one hand, and adaptability and variability on the other hand.

Avoid premature decisions

Important decisions (for example, with respect to requirements engineering or other activities in software development) are only made at the last possible moment. Until then, possible options are only observed and assessed. Decisions are only made when sufficient information about the problem is available.

Learn from failing early

At the beginning of a project, the degree of uncertainty is very high compared to later stages of the project. Therefore, decisions made in early phases of the project may prove to be wrong later when considering the insights gained during the development process. These kinds of errors must be accepted. If they happen early in the project and are also discovered early, these errors are not harmful, but help in eliminating uncertainty in the project.

Explore and adapt (instead of making predictions)

Whenever possible, uncertainties are eliminated by prototyping and conducting exploratory activities, but not by predictions. Exploration relies on analyzing an existing piece of software (or another kind of artifact), and then adapting the product or the process based on the newly gained knowledge. Making assumptions about future knowledge (i.e., making predictions), would be the opposite approach.

Expect change

In a project, late requirements or the late identification of changes that arise due to newly gained knowledge is to be expected. Therefore, the software process and the software system should be designed in a way that also allows for changes to the system to be incorporated into the project in later phases in a way that makes sense economically. A typical approach to this is to “make it work, make it right, make it fast” (i.e., to concentrate on functional requirements first). In many cases, for example, performance optimizations

lead to a more complex system design which in turn makes changes and adaptations more difficult. Because of that, it often makes more sense to implement the required functions first, and then focus on the quality requirements.

This principle also applies to project documentation. Changing extensive, detailed, and complex specifications or detailed architecture descriptions is time-consuming. If a system is described as completely as possible at the very start of a project, and is based on assumptions, new insights that are gained during system development will lead to costly changes.

Balance predictive and adaptive work

Depending on the project constraints, the planning up-front (the "prediction") is conducted to a reasonable extent. This comprises the elicitation of requirements, writing of the specification, and agreeing on a project plan. Influencing factors include the following:

- type of product
- the degree of end, mean, stakeholder, and domain uncertainty
- regulatory and legal requirements
- technical, domain-specific, and process-specific constraints

The extent of planning up-front should be weighed against the possibility of making changes and adaptations during the development process while still considering project constraints.

Validated Learning

This category contains principles related to quickly acquiring knowledge about whether assumptions made are correct or incorrect.

Validate important assumptions as quickly as possible

Many planning and design activities are based on assumptions, such as how the users will use the system or how the system will behave under high load. However, unconfirmed assumptions increase the project risk. Therefore, the development process should be designed in a way that important, and thus critical, assumptions are validated or falsified as quickly as possible.

Aim for fast feedback

Obtaining feedback on whether the requirements have been understood and implemented according to the clients' expectations, or whether the system components behave as originally planned, is an essential element of gaining knowledge in the software process. Therefore, the software development activities should be structured in a way that allows for quick feedback on decisions and assumptions made. This applies to both the created artifacts and the organization of the development process. An example for realizing this principle is prototyping.

Work in Progress (WIP)

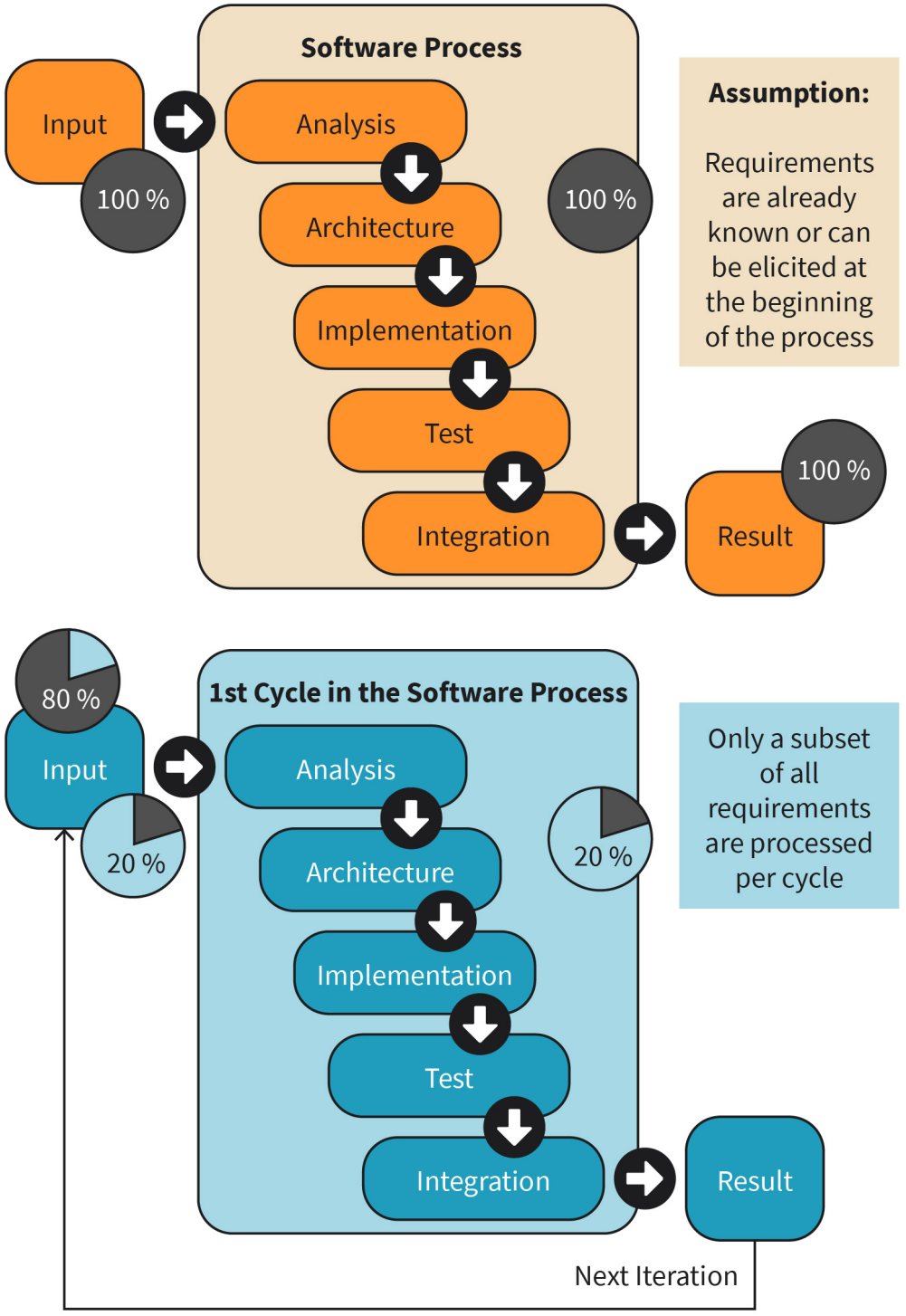
The term work in progress (WIP) refers to ongoing work that is not yet fully completed. Thus, it can be used as a measure to quantify ongoing work packages or artifacts in progress. The following principles refer to the organization of WIP in software projects.

Consider smaller batch sizes

Batch size is a measure for the proportion of work packages that are processed per work step (also phase, cycle, iteration, and sprint) in the software process. For example, a batch size of 100 percent means that all requirements are specified in detail in one go. After that, the design for 100 percent of the requirements is created, then 100 percent of the requirements are implemented, tested, and finally put into operation. This approach is realized in the waterfall model (as illustrated in the figure below). A batch size smaller than 100 percent means that the further activities in the process are performed before the current batch of requirements, features, or tasks are completely processed. An extremely defined batch size of 1 means that only one requirement is specified at a time, then implemented, tested, and put into operation. Small batch sizes have the following advantages (Rubin, 2013, p. 49):

- reduced cycle time: Small batches of work can be processed more quickly through the entire cycle than larger batches. This shortens the time between starting work on a requirement and completing its implementation.
- faster feedback: Due to the reduced cycle time, faster feedback is possible, which in turn leads to faster elimination of uncertainties.
- reduced risk: By reducing the batch size, the risk for a failing implementation is also reduced for this cycle because there is only a small number of requirements that are to be implemented. If there are problems with the ongoing implementation, these are limited to the current cycle. Although the problems might impact the overall project plan, the impact is limited to changes to the plan. For example, since there is only a subset of all requirements that are selected for implementation in one cycle, there is no need to rework the entire set of fully specified requirements.
- reduced cycle complexity: Smaller batch sizes mean fewer tasks which again, has an impact on the cycle complexity. There are fewer domain-specific and technical dependencies that must be considered in one cycle. This also reduces the organizational and coordination efforts between the team members.

Figure 8: Comparing Projects With Different Batch Sizes (Top: 100 %, Bottom: 20 %)



Source: Created on behalf of the IU (2023).

Manage the inventory of knowledge assets

Inventories are critical indicators in the manufacturing or trading sector. Although, unlike in the manufacturing sector, software projects do not have warehouses, the amount of started and not yet completed work (WIP) can be considered the current inventory. A requirement that has already been specified in detail but has not yet been implemented, can be considered a stock item of the knowledge base. A detailed specification consumes resources. If the specification is not yet implemented, the resources consumed have not yet contributed to the result visible for the client. The higher the inventory, the higher the effort for adaptation, if the basic assumptions or existing requirements change during the project.

Imagine the following situation: a plumber is awarded a contract for equipping all 300 rooms in a hotel with new faucets. Because they have resources available and they have a good offer on faucets, they build up a stock of 300. By doing so, they ensure that their employees will always have access to a sufficient stock of faucets. After the first 120 rooms have been converted, more and more complaints are issued by the hotel guests because the new faucets can no longer be operated with one hand. As a result, all rooms must now be equipped with different faucets.

In this example, a very large stock was built up (large batch size), user feedback was only considered at a late stage (late feedback), and the stock items lost value or had to be reworked (wasted resources). Although this metaphor cannot be applied to software processes one-to-one, it illustrates the risks of a large amount of WIP. Writing a comprehensive specification at the beginning of a software project represents a big inventory of knowledge work. This implies a high risk of change and adaptation if important assumptions that are made during the specification process turn out to be wrong later. Therefore, an efficient inventory management aims at continuously observing the WIP and balancing between batch sizes that are either too large or too small for a project or a project phase.

Ensure the flow of work

In situations where there are many cycles with relatively small batch sizes it is important that the roles and people involved in the process work together efficiently and effectively. Delays in the workflow (bottlenecks) lead to increased WIP, which corresponds figuratively to an increase in stock. The principle “ensure the flow of work” aims at avoiding unnecessary WIP and achieving the best possible flow in the processing of all phases and phase transitions of a requirement. A frequent error in process optimizations like this is looking for employees with incomplete workloads and increasing their work assignments accordingly, instead of identifying critical bottlenecks. In the worst case, this leads to even more open tasks in the delayed workflow than can actually be reduced.

Calculate the cost of delay

When creating project plans and managing the workloads of employees, possible costs of delay must be considered. It might be better not to assign the full workload to your employees over the complete project duration in order to avoid excessively full workloads

over a short period of time at the end of the project. A postponed delivery date can significantly increase the risk of cost of delay. Accordingly, from an economic perspective, it makes more sense to distribute activity over the whole project period.

The following example illustrates this principle (Rubin, 2013, p. 53): In a project, the documentation of the implemented software system is defined as a necessary acceptance criterion. If the documentation is created by a team member at 100 percent of their workload at the end of the project, this will increase the risk of cost of delay. Conversely, if the documentation is created by a team member throughout the project at a workload of 70 percent (which is not their full workload), this will reduce the risk of cost of delay. This approach might make more sense from an economic perspective if the expected cost of delay is very high.

Progress

This category is comprised of principles that address the measurement and control of the project progress.

Continuously deliver value to the customer

In agile projects, the only thing that counts is the value created for the customer. Although the artifacts created in a software project are relevant for the communication and the organization within the project, these artifacts usually do not represent directly usable value to the customer. Therefore, agile projects focus on the continuous delivery of functions that add value to the customer's business purpose.

Only working, validated assets deliver value

In agile software engineering, project progress is not measured based on a planned status but based on the created artifacts. In the case of executable software, this is can be the number of implemented requirements or the value that the software creates for the business. Only the goods that are delivered to the customer and their quality characteristics are considered a result of the project. The amount of WIP (i.e. work that has been started but not yet finished) is explicitly not considered.

Measure progress continuously and adapt planning

The progress of the project is continuously measured and the project plan adapted and changed accordingly. Instead of adhering strictly to the original plan, the goal is to adapt the plan appropriately based on the new knowledge and feedback obtained.

Performance

This category includes principles that address team performance and performance management.

Go at a sustainable pace

Agile projects aim at achieving the fastest possible development speed without overburdening the team. The goal is to establish a continuous velocity that can be maintained by the team in the long term. Extra sessions at night or over the weekend, can provide short-term productivity increases, but the subsequent recovery phase could lead to a drop in average performance.

Continuously deliver high quality

In agile software engineering, quality assurance activities are performed throughout the project, not only at the end of the development phase. The aim of continuous, adaptive quality management is to avoid unnecessary errors (for example, those caused by insufficient test coverage). Note that in agile software engineering, the whole project team is responsible for the quality of the product, not just the testing department.

Avoid unnecessary formalities

In agile software engineering, the focus is on delivering working software that represents value to the customer. This also applies to effective documentation. In agile projects, documents are generally only created if they add actual value or if they must be created due to strict constraints such as legal requirements. All unnecessary documents or management artifacts – including the tasks needed to produce these artifacts – should be omitted. These principles can be used as a set of guidelines for designing decision-making software development processes. They can also form the basis for an analysis of the degree to which an organization does or does not practice agile software engineering.

SUMMARY

The exact costs, functions required, and technical design of a software system can only be fully determined at the end of the project. Thus, industrial software processes involve making decisions in the face of uncertainty and ignorance. Relevant types of uncertainty include end uncertainty, mean uncertainty, stakeholder uncertainty, and domain uncertainty.

In contrast with classic software development, agile software development aims at making all important decisions based on the knowledge gained during the development process. This approach is called “knowledge-driven” software development.

The principles of agile software development provide an organizational strategy that guides the design of concrete and individual software processes. These principles can be structured according to the categories of variability and uncertainty, prediction and adaptability, validated learning, work in progress (WIP), progress, and performance.

UNIT 2

AGILITY IN SMALL TEAMS WITH SCRUM

STUDY GOALS

On completion of this unit, you will be able to ...

- understand the general setup, process, and roles of Scrum.
- explain how the product backlog is used in Scrum
- describe the product backlog characteristics and core elements.
- identify which management artifacts should be employed in a Scrum project.

2. AGILITY IN SMALL TEAMS WITH SCRUM

Introduction

In this unit, you will learn how to use Scrum as a process framework for agile software engineering in small teams. First, we will look at the various roles and activities involved in the Scrum framework. Then, we will move on to a detailed look at the product backlog - the central management artifact used in Scrum. Finally, we will explore other typical management artifacts that are frequently used in Scrum projects.

2.1 Basics of Scrum

Scrum is a highly evolutionary software process model framework that is characterized by organization in short cycles and by self-organizing teams. Interestingly, Scrum does not involve a definition of any roles or activities that are specific to software engineering. Therefore, Scrum is not really a specific software process model framework but instead, can be applied as a process model framework in different disciplines. Scrum defines various roles, strictly prescribes activities and the order in which they are executed, and introduces specific elements for managing projects.

The Scrum Framework

Product owner:
The product owner is the role in Scrum that represents the customer and that is responsible for the product's success.

In Scrum, there are three key roles: the product owner, the scrum master, and the team. The **product owner** represents the needs of the customer or the user. In a Scrum project, they are responsible both for the product's success and for requirements engineering (RE). The product owner forms the bridge between the project and the outside world because they are the only interface between the stakeholders and the developers of the system. Important domain-specific decisions are made by the product owner in Scrum. The product owner creates the release plan, prioritizes requirements, and accepts the result that has been built in a development cycle.

Scrum master
The scrum master is the role in Scrum that facilitates all activities and ensures that the team can work effectively.

The **scrum master**is an organizational project manager. They moderate and support the activities in Scrum and make sure that all stakeholders meet the given timelines and organizational constraints in a Scrum cycle. Furthermore, the scrum master is responsible for setting up an ideal work environment for the product owner and the team by resolving any impediments that prevent the team from working effectively. However, the scrum master has no authority over the product owner or the team. They are responsible for managing the Scrum process by enabling the development team and the product owner to do their work and protecting them from external influences or disturbances.

The team is responsible for the technical design, the construction, and quality assurance of the software. Scrum teams are self-organized and autonomous. In coordination with the product owner, the team decides how many functions are implemented and when they are implemented in a cycle. Apart from organizational specifications and conven-

tions, the team decides on the technical implementation. All roles that are involved in software development (for example, architect, developer, and tester), work closely together in one team. A Scrum team is typically comprised of three to nine members. At the end of a cycle, the team presents the result they have achieved.

Elements for Managing and Controlling the Project

In a Scrum process, the most important concepts are the product backlog, the sprint backlog, and velocity. The **product backlog** comprises a list of the requirements for the system. The requirements can vary in detail – ranging from initial ideas to fully specified functions. At the beginning of a software project, the product backlog often contains only a few, roughly outlined requirements for the system. During the process, the product owner refines, supplements, and prioritizes the requirements. Only the product owner is allowed to make changes to the product backlog.

Product backlog
The product backlog is a collection of all requirements for the system.

The **sprint backlog** comprises a list of requirements that are to be implemented by the team in the next development cycle, the so-called sprint. The sprint backlog is a subset of the product backlog. The product owner and the team agree which requirements will be included in the spring backlog. By doing so, they also determine the batch-size of the sprint. The batch-size depends on the velocity of the team. During the ongoing sprint, the sprint backlog must not be changed.

Sprint backlog
The sprint backlog is a list of requirements that are to be implemented in a sprint.

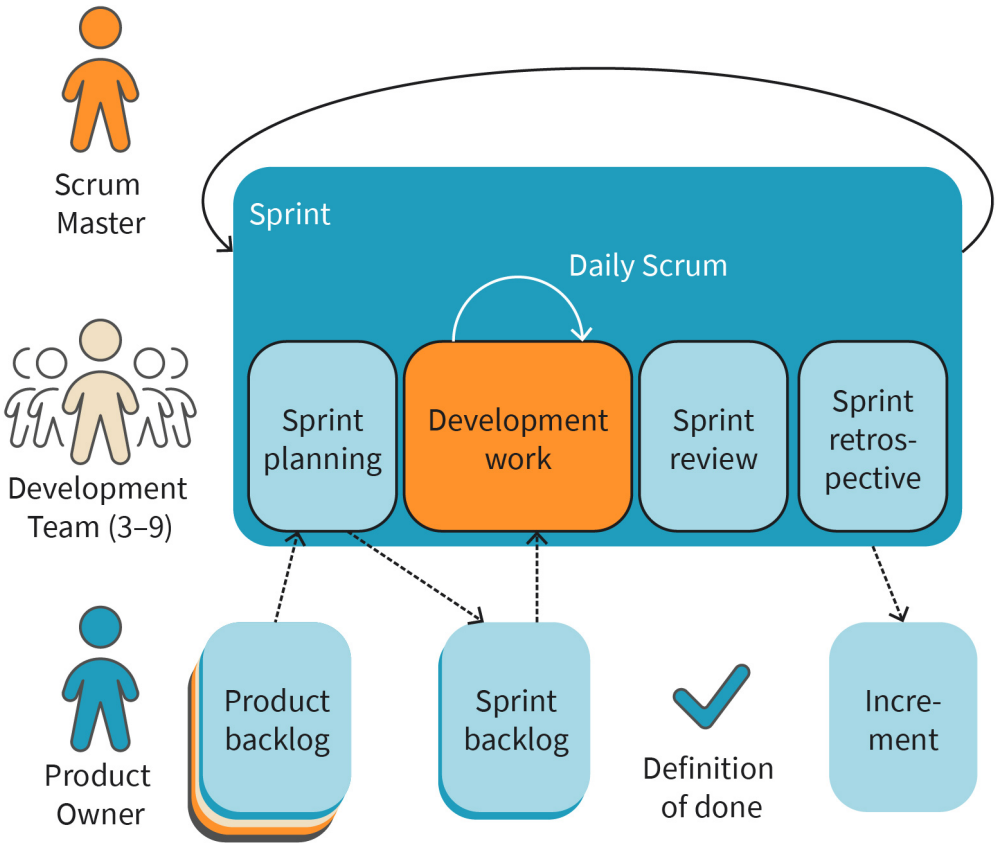
The concept of **velocity** is not specified as part of the Scrum framework but nevertheless it is frequently used in Scrum projects. The velocity is a team-specific measure that indicates the quantity and scope of functions the team completes per cycle. Typically, the velocity of a team stabilizes after the first five to seven cycles. Based on the velocity, the team decides how many and which requirements they can select from the product backlog and include in the next sprint backlog.

Velocity
The velocity is a team-specific measure that indicates the work a team completes within a cycle.

Scrum Activities

The figure below shows the overall structure of a Scrum process. For every sprint, the sprint backlog is created from the product backlog. After that, the team implements the requirements from the sprint backlog in the sprint. A sprint has a fixed length, such as 30 days. Every day, there is a short meeting to present what was done in the last 24 hours, discuss any problems that occurred, and decide what will be done in the next 24 hours. This daily meeting is called “daily scrum” or “stand-up meeting” and it is strictly time-boxed to 15 minutes. At the end of the sprint, the team presents the extended piece of software and compiles the list of requirements for the next cycle.

Figure 9: Scrum



Source: Created for the IU (2023), based on Kneuper (2018, p. 104).

Activities and Events of a Sprint

Scrum defines the phases of a cycle and the roles and the activities that these roles execute exactly. A sprint consists of the following events (see also the figure above):

- sprint planning
- development work
- daily scrum
- sprint review
- sprint retrospective

In the sprint planning phase, the product owner presents the requirements they have selected for the current sprint. The team then estimates the effort needed to implement each of the requirements. After that, the sprint backlog is filled: the team chooses the requirements the product owner has assigned top priority and limits the number of requirements so that the overall effort needed to implement them in the sprint backlog matches the team's velocity. Finally, the team discusses how they will implement the requirements and creates an ordered work plan with tasks assigned to every team member.

During the sprint, the team implements the requirements from the sprint backlog. The product owner is available for any questions that may arise. During the sprint, however, the main activity of the product owner is to specify the requirements for the next sprint (cycle). Normally, the number of requirements in a sprint remains stable, but can also be adjusted if necessary. This must be agreed upon by the product owner and the development team. Adjustments may be needed if the requirements cannot be implemented in the time frame of the current sprint. Because the duration of a sprint is fixed, it is only the contents of the sprint that can be adjusted if necessary (commonly known as “time-boxing”). The scrum master resolves impediments during the sprint and is responsible for enacting the daily stand-up meetings.

In the sprint review, the team presents their result to the stakeholders and obtains feedback. The product owner officially accepts the sprint result and documents any changes or requirements that are not implemented yet in the product backlog. A sprint ends with the sprint retrospective. In this meeting, the team reflects on the current sprint process and discusses improvements for the future. You can think of it as a “lessons learned” session as is widely used in classic software engineering.

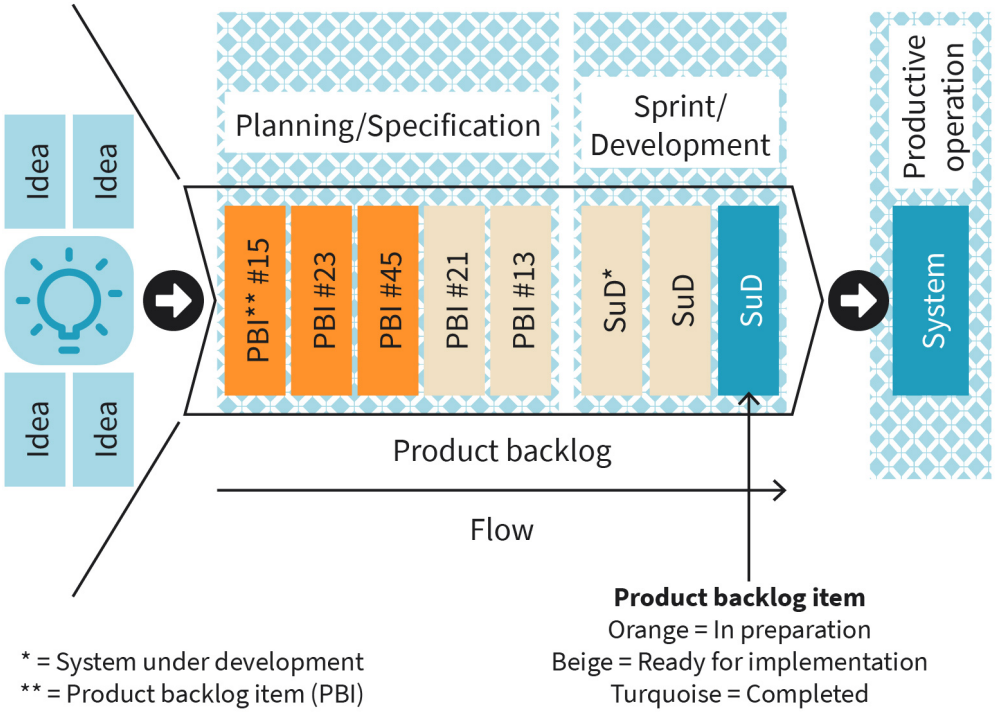
2.2 Central Management Artifact: Product Backlog

The product backlog is one of the most important management artifacts in Scrum. It is a prioritized list of functions intended for the product and activities that have to be conducted in order to achieve the project outcome. The product backlog is never complete, stable, or finished. It is a management artifact that reflects the current state of knowledge at a given point in time.

Pipeline Concept and Flow Management

As shown in the figure below, the product backlog can basically be thought of as a pipeline. The elements that are transported in this pipeline are called "backlog items". Backlog items contain knowledge and insights that the development team needs to create relevant product artifacts for the customer. Once an item is fully implemented, it can flow out of the pipeline. It then delivers added value to the customer.

Figure 10: Product Backlog as Pipeline



Source: Created on behalf of the IU (2023).

The last consumer in the pipeline is the development team. In the sprints, the team works on the items that are ready for implementation. An item that was included in the product backlog as an initial idea must be prepared for implementation. This means that the team must be provided with enough information so that they are able to create the program code. This information must be elicited up front and documented appropriately.

If the flow of backlog items through the pipeline is interrupted, for example, due to an impediment, the whole process can be subject to delays and failures. That is why Scrum aims at enabling a steady flow through the pipeline, continuously creating added value for the customer. This serves as the basis for defining roles and activities in Scrum.

Scrum does not focus on the workload of individual team members, but on the efficient, unhindered flow of items through the backlog. The scope of the pipeline is scaled according to the current perspective. It can be used for planning the product, the release, or the sprint level. The basic concepts always stay the same. The goal is the orderly, adaptive planning and implementation of the resulting artifacts based on the current state of knowledge. By continuously and actively maintaining the backlog items, the knowledge gained by delivering interim project results can be incorporated and considered in the ongoing process.

Types of Product Backlog Items

Basically all requirements for the systems are collected and processed in the backlog. However, the team does not only implement new requirements. Thus, Rubin (2013, p. 101) proposes the following types of product backlog items:

- **"add functions"** items involve implementing and adding new functions required by the customer to the system.
- **"change a function"** items involve changing an existing function of the system as requested by the customer.
- **"defects"** items involve fixing known system errors.
- **"technical work"** items improve the system technically. The improvements do not directly reduce errors, but increase the quality of the system (e.g. maintainability)
- **"acquire knowledge"** items aim at eliminating uncertainty but are not directly visible in the management or product artifact

These different types of product backlog items allow the product owner to not only manage new functions, but also to work on items that form part of the development process and that do not directly manifest themselves in system functions.

Attributes of Product Backlog Items

Since the product backlog items are refined and managed from the idea to the implementation, every item is described by a set of metadata that must be collected and stored. Röpstorff & Wiechmann (2012) propose to use the following attributes:

- unique identifier (ID)
- description, user story: describing the contents of the backlog item, typically in the format of user stories (Cohn, 2022)
- acceptance criteria: a set of conditions that must be met to formally mark the item as completed.
- management attributes: information required for managing and planning, for example, value contribution, priority, relevance, implementation risk
- complexity, estimation results: indicators for the complexity of implementation, or estimation results for the implementation effort
- category (grouping): assigning a higher-level feature category to the user story or other type of requirements description, for example a use case or epic (see Hruschka et al., 2022).
- notes: links to detailed requirements artifacts, remarks, comments, or related backlog items

When setting up a product backlog, the list of attributes can be individually adapted depending on the type and requirements of the project at hand. The product owner selects the suitable attributes based on the given constraints and documentation rules. They can be changed during the project and can also be varied according to the underlying item type.

Quality Characteristics of the Product Backlog

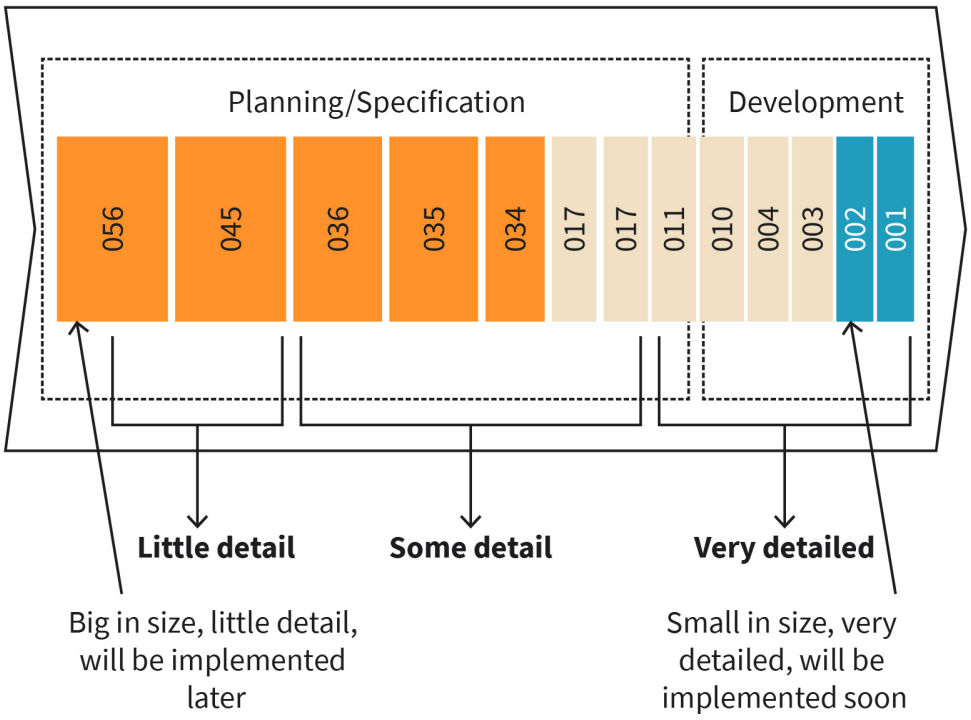
In order to assess whether a backlog is well structured or of sufficient quality, the backlog or backlog items should be checked to ensure that they are (Cohn, 2009; Rubin, 2013):

- detailed appropriately
- emergent
- estimated
- prioritized

A product backlog is considered to be detailed appropriately if the level of detail of its elements is such that all stakeholders are able to do their work on this basis. The figure below illustrates this characteristic. The given requirements should be implemented in a Scrum sprint. Therefore, all information that is needed for implementation must be available. When backlog items are described in this level of detail it is also called “enabling specification”, because it enables the developers to do the implementation work of a sprint without further information from the product owner (Sutherland & Coplien, 2019. Based on this, we can derive the following findings in relation to the product backlog:

- The team should be on the 0^{th} Level of Ignorance with respect to the backlog items that are prepared for implementation (enabling backlog items).
- Enabling backlog items should expose a relatively high degree of detail (compared to other items) and should be relatively small in size.
- Recently added backlog items that are to be implemented later are generally roughly defined (i.e. they are less detailed and bigger in size).
- As the project advances, the relevant backlog items must be refined and broken down into smaller pieces as appropriate. Here, it is important to find the right balance between continuously providing the necessary information to the development team and not creating an unnecessarily high stock of knowledge.

Figure 11: Different Degrees of Detail in the Product Backlog

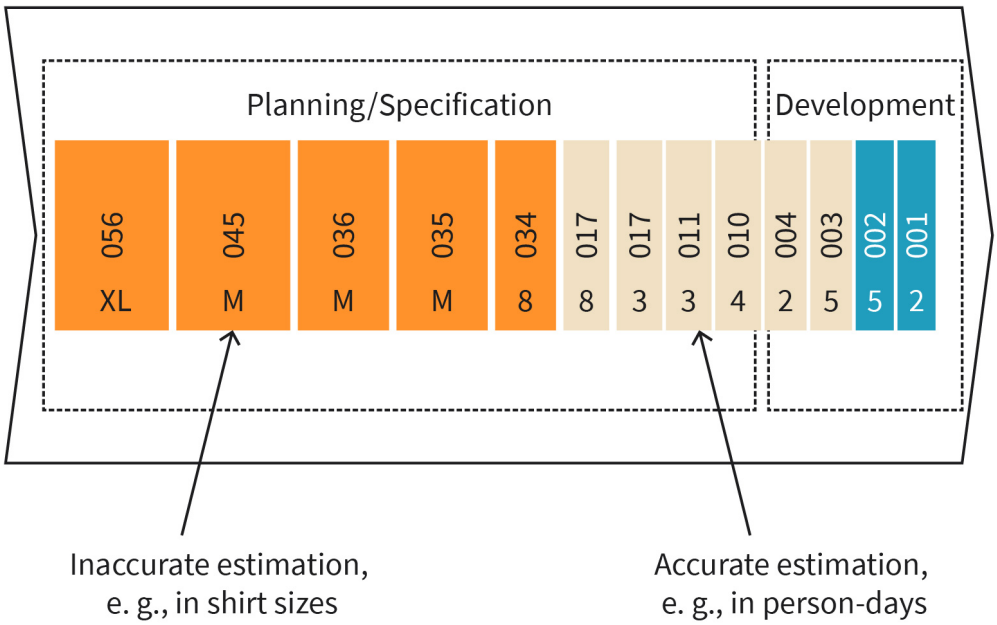


Source: Created on behalf of the IU (2023).

The backlog items should be prioritized at any point in time. Independent of the level of detail, the order of the product backlog items should always correspond to the currently planned state of the implementation. Taking the figure above as an example, this implies that the items are consistently processed from right to left. High-priority items must be located at the very front (or on the right or depending on the type of visualization at the top) in the product backlog. Less important items or items that are classified as irrelevant are located at the back (or on the left or at the bottom).

All items in the product backlog are estimated according to the current level of knowledge. Since new items are typically of little detail and bigger in size, they cannot be estimated as accurately as already more detailed and smaller items. The figure below shows an example of a product backlog containing items that were estimated partly accurately and partly inaccurately. One way of intentionally making inaccurate estimations is to estimate based on shirt sizes (see the figure below). By using shirt sizes such as XS, S, M, L, XL, and XXL, backlog items can be roughly categorized. As soon as enough details are available for a backlog item (later on in the project), a meaningful estimation based on person-days or other more precise estimation units can be made. Then, the estimation metric is adjusted accordingly.

Figure 12: Estimations in the Product Backlog



Source: Created on behalf of the IU (2023).

Three of the main characteristics of a product backlog – detailed appropriately, prioritized, and estimated – can be assessed at any time. However there is another key characteristic that can be used to describe a product backlog overall: emergence. The concept of emergence is defined as "the fact of […] becoming known or starting to exist" (Oxford University Press, 2022). A product backlog is said to be emergent if it reflects the current state of knowledge, but is also continuously updated by changing and adapting items and their attributes based on new insights. Therefore, the product backlog changes constantly in the course of the project. Typically, the following types of changes occur in emergent product backlogs:

- adding items
- removing items or grouping them into one item
- splitting items into several more detailed items
- reprioritizing, re-estimating, or adapting items in size

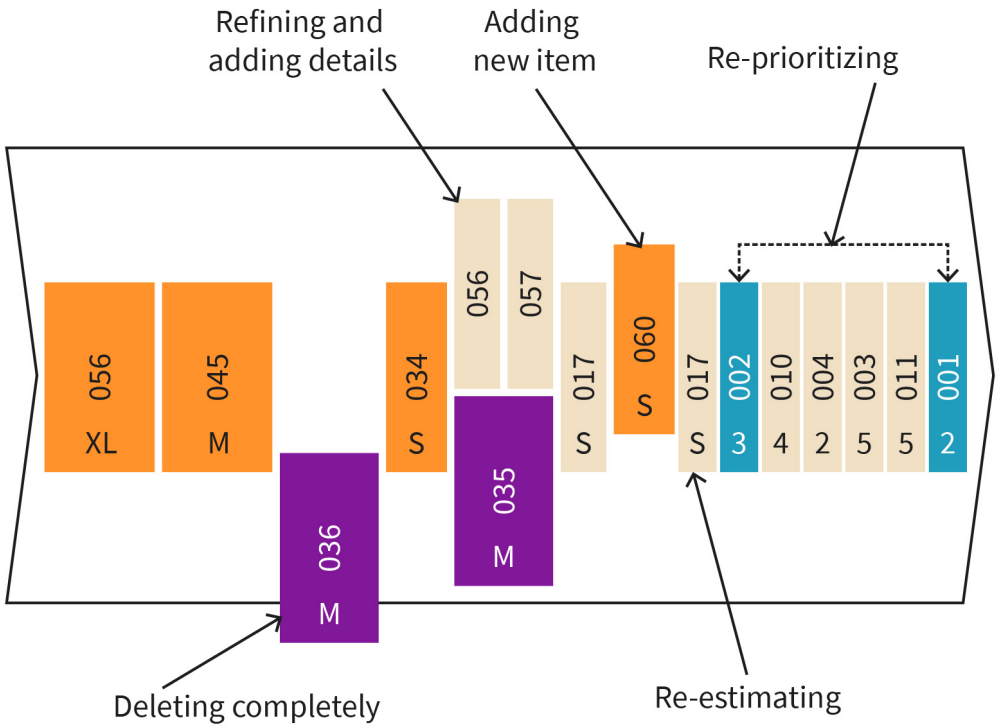
As a consequence, a correctly used product backlog is never finished, nor does it reach its final state. If a product backlog does not show the characteristic of being emergent, it is a strong indicator that the project does not apply agile software engineering. In agile software engineering, project control based on feedback is a key element, which in turn is reflected in the fact that the product backlog is emergent.

Backlog Refinement: Continuously Maintaining the Product Backlog

The product backlog is the central management artifact in Scrum. Important decisions regarding project control, as well as decisions and findings regarding the implementation, are stored in this artifact. Therefore, an important characteristic of the product backlog is

continuously adapting and changing its contents. It is the product owner, who is mainly responsible for maintaining and adapting the product backlog. The actual work is normally done in close collaboration with the project stakeholders. The figure below illustrates typical activities involved in maintaining the product backlog.

Figure 13: Activities for Maintaining the Product Backlog



Source: Created on behalf of the IU (2023).

When re-prioritizing items, their order is changed. When re-estimating the effort for an item, the size of the element is adapted. New items can be inserted at any position. Existing items can be refined, detailed, or deleted completely. Furthermore, all other attributes of a backlog item can be changed.

2.3 Other Management Artifacts and Tools

The most important management artifact in a Scrum process is the product backlog. Other Scrum artifacts are the sprint backlog and the product increment as the central result of a sprint. In addition, there are various other management tools that are frequently used in Scrum:

- definition of done (the sole element in this list that is explicitly defined in the Scrum framework)

- definition of ready
- task board
- velocity;
- burndown chart.

Sprint Backlog

As mentioned above, the sprint backlog is a subset of the product backlog items. During the sprint planning, the product owner and the team jointly select a set of appropriately detailed and prioritized items from the product backlog for implementation in the next cycle. The number and size of the items in a sprint backlog are determined by the velocity of the team. The team decides how many items they can safely implement within a cycle and thus, shall be included in the sprint backlog. During the cycle, the number of sprint backlog items must not be changed (i.e. during the sprint execution no other items from the product backlog are added to the sprint backlog).

Definition of Ready

Before a product backlog item can be implemented, it must meet the "definition of ready" (Sutherland & Coplien, 2019, p. 316) according to a list of criteria (usually in the form of a checklist). This checklist is specific to each project and agreed upon at the project start. Only items that adhere to the definition of ready can be included in the sprint backlog. The entries of the checklist represent completed tasks. The following table provides an example of a definition of ready checklist.

Table 3: Example of a Definition of Ready Checklist

Checklist: Definition of ready for product backlog items that are ready for discussion in the sprint planning meeting	
☐	The business value of the planned deliverable is clearly stated and understood.
☐	The team has understood all the item's details. They are able to decide whether they can successfully complete the item.
☐	All dependencies have been identified. There are no internal or external dependencies that prevent the team from completing the item.
☐	The team has enough resources to complete the item.
☐	The team has estimated the item and can complete it within the sprint.
☐	The acceptance criteria for the item are known and testable.
☐	If there are special quality criteria, these are known and testable.
☐	The team knows how to demo the completed item in the sprint review meeting.

Source: Created for the IU (2023) based on Rubin (2013, p. 109).

This checklist represents a kind of quality gate for product backlog items, which can only be passed if the item meets all criteria.

Table 4: Example of a Definition of Done Checklist

Checklist: Definition of done for completed product backlog items	
☐	Design is reviewed.
☐	Code is implemented.
☐	Coding conventions are respected.
☐	Code is under version control.
☐	User documentation is up to date.
☐	All unit and module tests passed successfully.
☐	All acceptance criteria are met.
☐	User acceptance tests have been successfully conducted.
☐	No known defects.
☐	In operation.

Source: Created for the IU (2023) based on Rubin (2013, p. 74).

The definition of done is also a quality gate that can only be passed if the item meets all criteria. The consistent application of such checklists in practical software engineering ensures a high process quality. This is because developers (especially in stressful situations such as approaching deadlines) often consider quality assurance or documentation activities as tedious work that does not contribute any new features to the system.

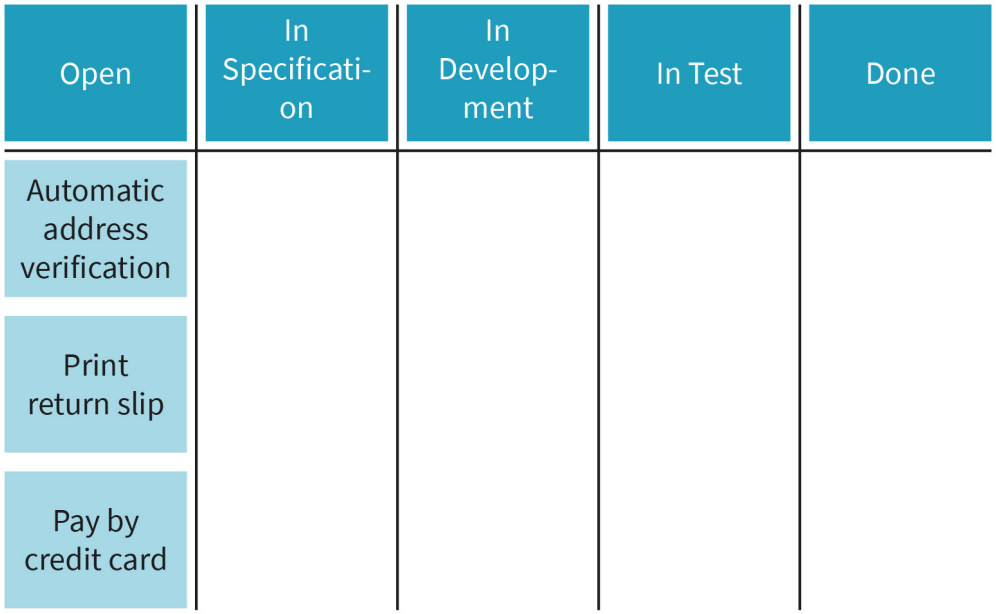
Task Board

A task board is a simple tool for visualizing all tasks of a sprint along with their current completion status. A special variation of the task board is the Kanban board. Using Kanban boards is a technique that originated in the automobile industry, namely at the car manufacturer Toyota Production Systems, who introduced Kanban boards to control their production processes (Monden, 1994). A Kanban card (Kanban means “visual sign” or “visual card” in Japanese) contains the type and quantity of an interim product that is soon needed in the overall manufacturing process and that must be produced shortly. Using Kanban cards, the upstream stations within the manufacturing process know what will be requested soon and can make sure that everything is prepared and available when needed. The cards are put on the wall or on special boards (called Kanban boards). This fundamental principle can also be applied for visualizing workflows within an IT project.

All tasks of a sprint are written on cards and placed in the far-left column of the Kanban board (Anderson, 2011, p. 73). The underlying period can be a week, a phase of the project, or if applied within the framework of evolutionary software development, a cycle. The figure below shows what a Kanban board might look like. This Kanban board contains three functions that are to be implemented for an online shop. As soon as the team starts

to specify one of these functions, the corresponding card is moved to the column “In Specification.” As soon as the team starts implementing the function, the corresponding card is moved to the column “In Development.”

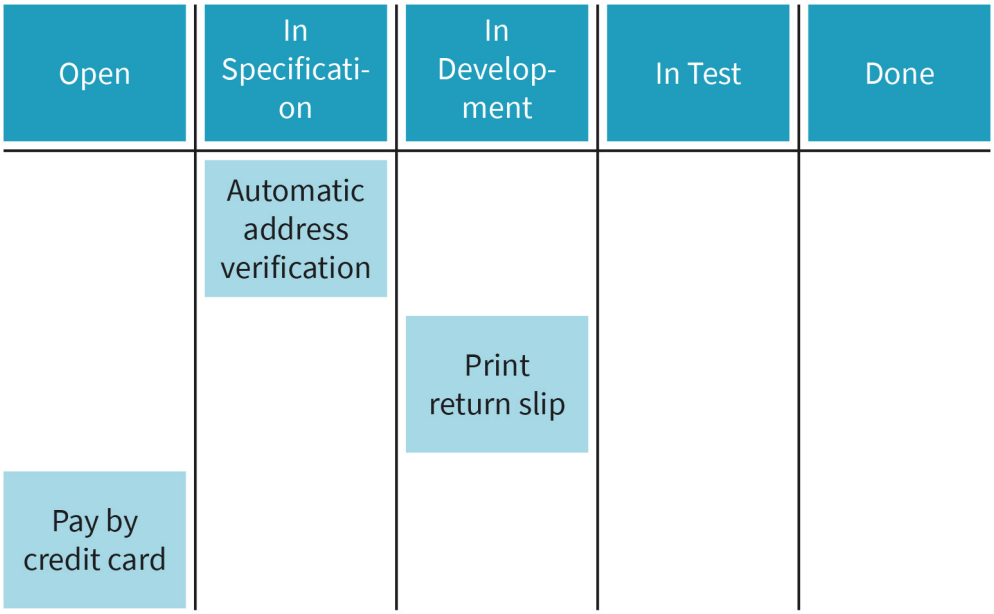
Figure 14: Kanban Board, Start of Process



Source: Created on behalf of the IU (2023).

If you look at the Kanban board in the figure below, you can tell at first glance that the function “automatic address verification” is being specified and the function “print return slip” is being implemented at the moment.

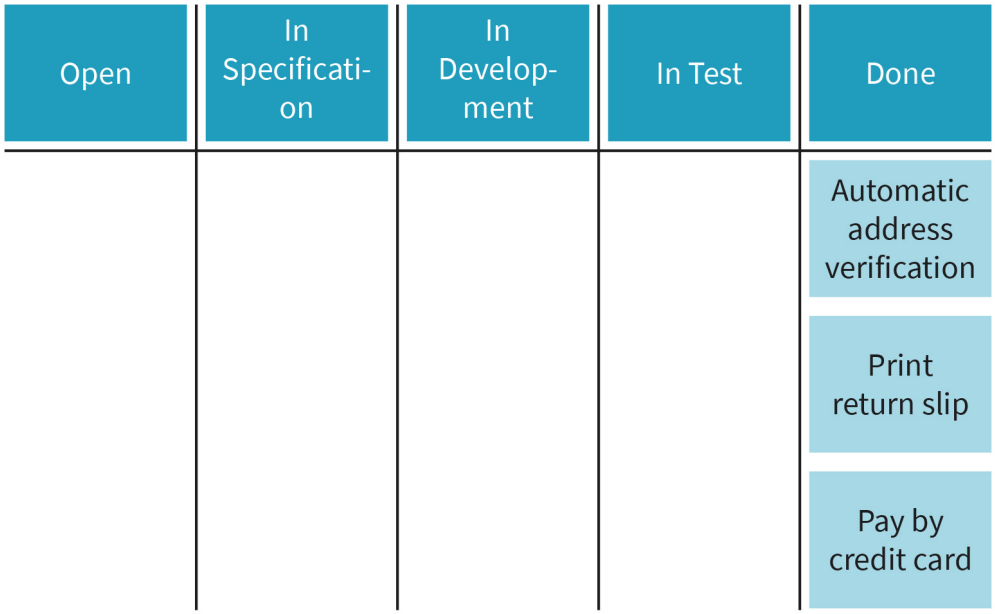
Figure 15: Kanban Board, Ongoing Process



Source: Created on behalf of the IU (2023).

The next figure shows what the Kanban board looks like when all tasks are completed. In the course of the work process, each card travels from left to right on the Kanban board. The number and names of the individual columns can vary depending on the project at hand. You can also add more information on the cards such as the assigned team member, estimated effort, priority, or other useful information.

Figure 16: Kanban Board, End of Process

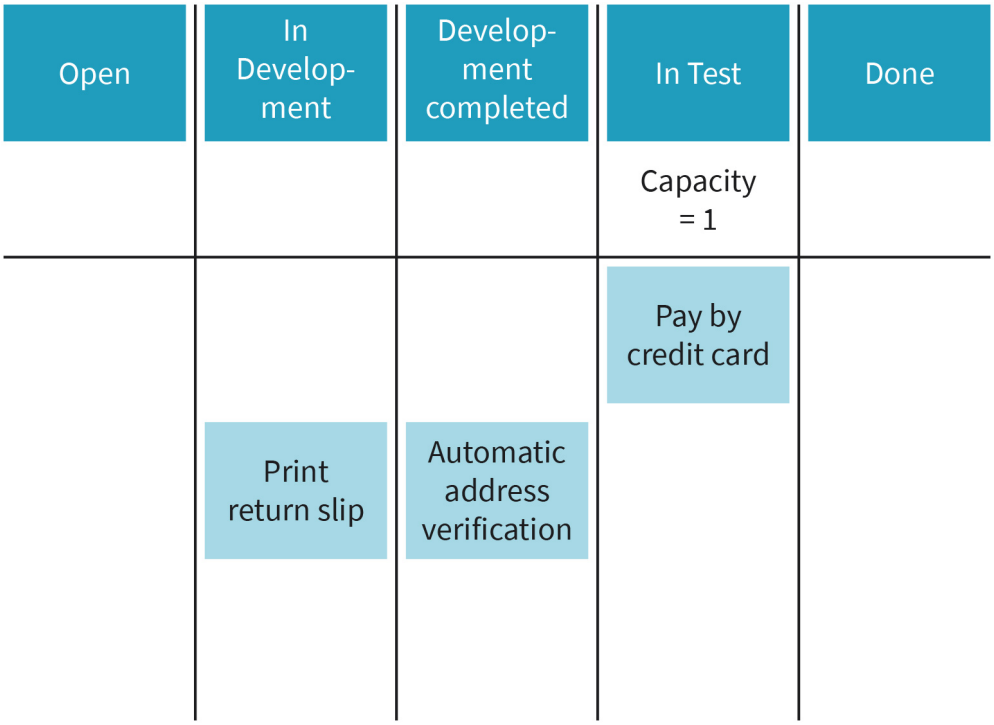


Source: Created on behalf of the IU (2023).

The figure below shows an extended Kanban Board. As opposed to the boards above, the Kanban board shown below uses different types of columns. While the boards above only differentiate between the different software engineering activities, the extended Kanban board distinguishes whether a function is being implemented or whether implementation is complete and the function is ready for testing. This allows you to see at first glance which functions are being worked on at the moment.

It is also apparent that only completed functions are ready to be moved to testing: in the case shown below, this applies to the function “automatic address verification.” In addition to more differentiated columns that distinguish between actual processing and being completed, the column “in test” of the extended Kanban board indicates a limit in capacity of 1. This means that only one function can have the status “in test” at any point in time. By using capacities, you can ensure, that, for example, every developer or every tester is assigned exactly one task. In the example shown below, the function “automatic address verification” is on hold until the function “payment per credit card” is readily tested.

Figure 17: Extended Kanban Board



Source: Created on behalf of the IU (2023).

Velocity

In Scrum, the term "velocity" represents a team-specific measure indicating how much work the team is able to complete per sprint (Sutherland & Coplien, 2019, p. 320). Velocity can be seen as a measure of the team's efficiency. Typically, velocity is described in story points or **person-days**, but there is no fixed prescription as to which unit of measurement shall be applied.

Person-days
This is an absolute unit of measurement used for indicating how long it takes to complete a task in terms of the number of days of work for a single person.

Story points are a relative and abstract measure used to relate the size of backlog items to each other. The more story points you assign to a backlog item, the more complex is the item and the higher is the estimated effort for implementing it. Using story points, statements like "the item is much bigger" or "the item is about the same size" can be made. Using abstract story points is based on the finding that people are better at making relative estimates than absolute estimates.

Story points
This is a relative unit of measurement used for indicating a team's velocity and estimating product backlog items.

After several sprints, you can determine the typical number of story points that the team is capable of achieving – provided the team composition remains the same. Only after a few iterations do the way of estimating and the speed of processing the tasks tend to converge. Therefore, in the first iterations the estimates are often imprecise but do stabilize over time. Thus, story points are a good basis for estimating the team's future performance and for assessing the amount of work that the team wants to take into a sprint.

Please note that story points cannot be transferred between teams, and thus, are not a measure for comparing teams. Take the example of team A having a velocity of 40 story points per sprint and team B having a velocity of 25. It may well be possible that the outcomes delivered by both teams are the same in terms of volume and complexity.

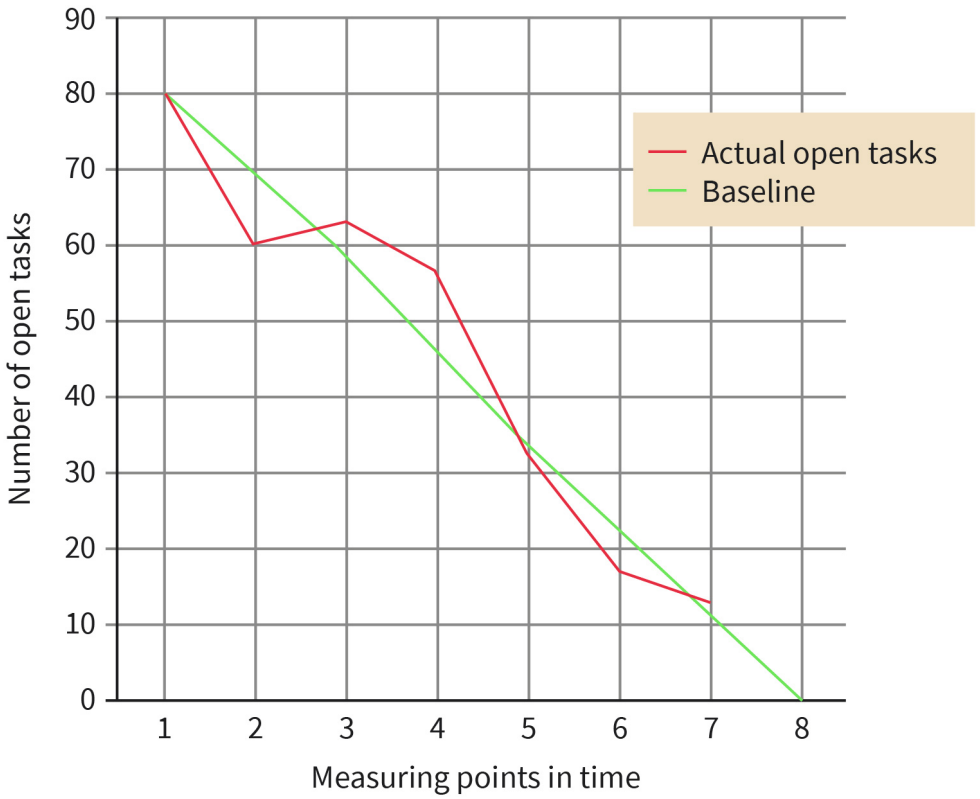
While story points are a relative estimation unit, using person-days for estimation is based on an absolute estimation unit. Estimating in person-days indicates how many working hours are required for finishing a backlog item. Sutherland & Coplien (2019) recommend using relative estimations (students can refer to Sutherland & Coplien, 2019, p. 320 for an in-depth discussion).

Burndown Charts

A burndown chart shows the current progress of a sprint based on the number of completed sprint backlog items in relation to the planned and available time. Basically, the term “burndown chart” refers to a class of visualizations that are used to illustrate aspects of the projects' progress.

The figure below shows an example of a burndown chart that illustrates the progress of open tasks in a work package over a period of eight points of measurement. Additionally, the burndown chart shows the baseline for the ideal number of open tasks over the depicted period of time.

Figure 18: Example of a Burndown Chart



Source: Created on behalf of the IU (2023).

The main characteristic of the burndown chart is the targeted curve from the top left to the right bottom of the diagram. Ideally, the actual curve meets the zero line at the end of the working period. Then, for example, all open tasks are “burned down.” This is how the visualization type got its name.

In a burndown chart, the X-axis (horizontal axis) shows the chronological progression. The first value represents the starting point of the considered period. The period may comprise the whole project, a single work package, or a sprint in Scrum. The highest value of the X-axis denotes the end point of the period. In the example above, there are six intermediate measuring points between the sprint start (1) and the sprint end (8). The Y-axis (vertical axis) shows the unit of measurement that shall be plotted against the time. Usually, the number of open tasks or the number of used resources such as time or money are displayed on the Y-axis. In the example above, the Y-axis represents the number of open tasks, starting with 80 open tasks.

The baseline (the light-green line in the example) illustrates the ideal progress of the measuring points in the burn down chart. Usually, the baseline is linear and can be created by connecting the first measuring point at the sprint start (1) and the last point at the sprint end (8) using a straight line. The baseline’s intersection with the X-axis is at the

project end, since “burning down” all open tasks is the goal of the project. The actual course of the measured values is displayed in blue. This curve shows the number of open tasks that were completed in the current sprint.

By plotting the baseline and the measured values side by side, you can see at any point in time whether the sprint is progressing as planned or whether the team is advancing much quicker or much more slowly than expected. If the curve of the measured values lies above the baseline, this indicates slower progress. If the curve lies below the baseline, the sprint is advancing more quickly than planned.

SUMMARY

Scrum is a software process model framework that is organized in cycles. Scrum activities are conducted by the product owner, team, and scrum master. Scrum prescribes strictly what type of activities have to be performed and in which order. It also introduces Scrum-specific artifacts and tools for managing projects.

The product backlog is one of the most important management artifacts in Scrum because it reflects the current state of knowledge at any point in the project. The product backlog is a prioritized list of functions and activities that are agreed for the product or that must be conducted to achieve the targeted project outcome. Other frequently used management tools in Scrum are the sprint backlog, definition of ready, definition of done, task boards, velocity, and the burndown chart.

UNIT 3

AGILE PROJECT MANAGEMENT

STUDY GOALS

On completion of this unit, you will be able to ...

- name typical planning levels for agile projects and describe how they relate to each other.
- apply agile principles to portfolio planning.
- outline different approaches that can be used to organize and coordinate several teams in one agile project.
- explain how agile product and release planning is conducted.

3. AGILE PROJECT MANAGEMENT

Introduction

This unit introduces principles, methods, and techniques for agile project management on different planning levels. First, the individual planning levels and how they relate to each other are described. After introducing agile portfolio planning, the unit presents different approaches on how to organize several development teams in one overarching project. Finally, we will move on to an overview of how product and release planning can be carried out according to agile principles.

3.1 Planning Levels in Agile Project Management

In Scrum, the characteristics and principles of agility are (for the most part) directly applied and mapped to the roles and activities in a Scrum project. As a process model framework, Scrum organizes the work of small teams of three to nine people plus the product owner and the scrum master. However, industrial software projects are usually so complex that they are carried out by several dozen or hundreds of employees. Because of that, large IT projects must also be planned, organized and managed in a corporate context. Furthermore, the concrete software engineering activities that are carried out alongside the implementation must also be considered. If, for example, an important core system that is already in operation in an insurance company or a bank is going to be replaced by a new system, extensive preliminary work is necessary before the implementation can start. In summary, a suitable design of the software process should address the following challenges:

- integration of agile projects in a possibly non-agile enterprise context
- coordination and organization of large projects with several teams
- organization of project planning and control according to agile principles
- organization of software engineering activities according to agile principles

Consequences of Holistic Agility

What distinguishes agile principles and the reality in many businesses today is the planning and provision of resources for software projects. Project and resource planning is usually carried out based on detailed specifications. This, in turn, forms the basis for investment decisions and controlling.

The phrase “measure progress continuously and adapt planning” sums up one basic principle of agility and of Scrum in particular. According to this principle, you should not strive for an absolute adherence to the project plan created at the start of the project, but rather, adapt the plan appropriately based on the gained knowledge and the feedback received

during the project execution. Roughly speaking, the planning in Scrum is not done in a way that a certain product with certain features is finished by a certain point in time. Instead, human resources in the form of development teams are provided. After a certain period of time, the team delivers the version of the product developed up to that point, implemented based on the knowledge gained during the process.

Accordingly, in knowledge-driven projects, the uncertainties are explicitly highlighted, and the outcome is not predicted in detail. In contrast, in assumption-driven projects, very detailed assumptions are made and used as a basis for predicting a detailed outcome. In settings where both decision-making approaches meet, care must be taken to specifically coordinate the organizational interfaces.

Planning and Control Levels at a Glance

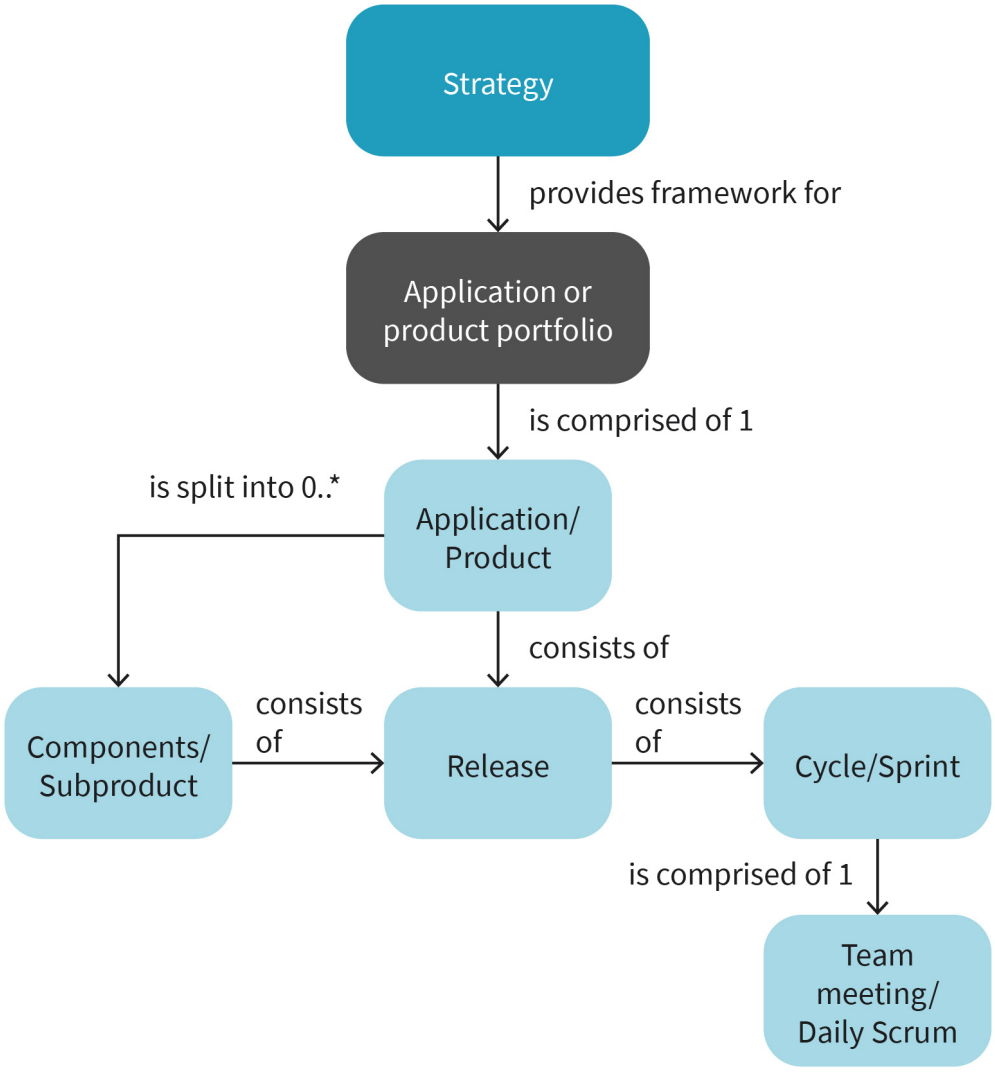
Generally, Scrum focuses on planning and working in a relatively small team. In the context of an enterprise, however, agile projects are part of the overall product planning. In the following section, we will focus on the possibilities and potential applications of agile (or knowledge-driven) project management. In particular, we will look at individual planning levels, from the **product portfolio** to the daily scrum. Typical planning levels of an organization include:

Product Portfolio: A product portfolio is the collection of products that a company works on to accomplish its business strategy.

- strategy
- application or product portfolio
- application/product
- Component/sub-product
- release
- cycle/sprint
- team meeting/daily scrum

This type of structuring is applicable both to organizations that create and distribute software products, and to organizations that utilize complex application landscapes for their business purposes. In the first case, the focus is on the product, in the second case on the application. The figure below shows the typical planning levels of an organization as they relate to Scrum.

Figure 19: Planning Levels



Source: Created on behalf of the IU (2023).

The strategy defines the basic orientation and framework conditions for the portfolio. Depending on the focus, this can be an IT strategy with fundamental **enterprise application management**(EAM) goals or a product strategy, if the development and distribution of software systems is the main focus.

Enterprise application management
This is the process of designing, managing and expanding the application landscape supporting the business processes of a company.

Decisions about which concrete applications or products are to be worked on, and in which sequence, are made at the level of the application or product portfolio. Resource allocation takes place at the level of portfolio planning. This includes deciding which resources are to be used for an application or a portfolio and for how long, taking into consideration the framework set out by the strategy.

As soon as an application or a product is included in the portfolio, it is actively worked on. A rough plan for development or implementation is conceived at the level of the application or product planning. Often, an independent project is initialized for this purpose. Larger applications or products that have longer development periods are typically broken down into smaller units, such as components or sub-products, each of which is organized and produced independently.

Accordingly, there is also a separate planning level for components or sub-products. Although, this planning level does not differ from the “application/product” level in structure, the outcome of this level does not stand alone; it is part of a larger application or product. Because of that, planning often has to take into account several dependencies on parallel projects or sub-projects in which other components or sub-products are being planned and built.

Delivery dates for new functions are planned at the release level. A release is a deliverable version of the application/product that can be used actively by the end-users/customers. You can plan releases both on the level of components and on the level of applications. Depending on the release frequency, these are typically medium to long-term plans that provide the basis for coordinating dependencies on other applications/products or components/sub-products.

At the cycle level (in Scrum, sprint level), the team members' concrete tasks are planned on the level of small work items (Sutherland & Coplien, 2019). In a sprint, value is created for the customer that usually manifests itself in new features integrated into the application. The team meets every day in a short team meeting (called a stand-up; in Scrum, daily scrum) to plan the daily work within a cycle.

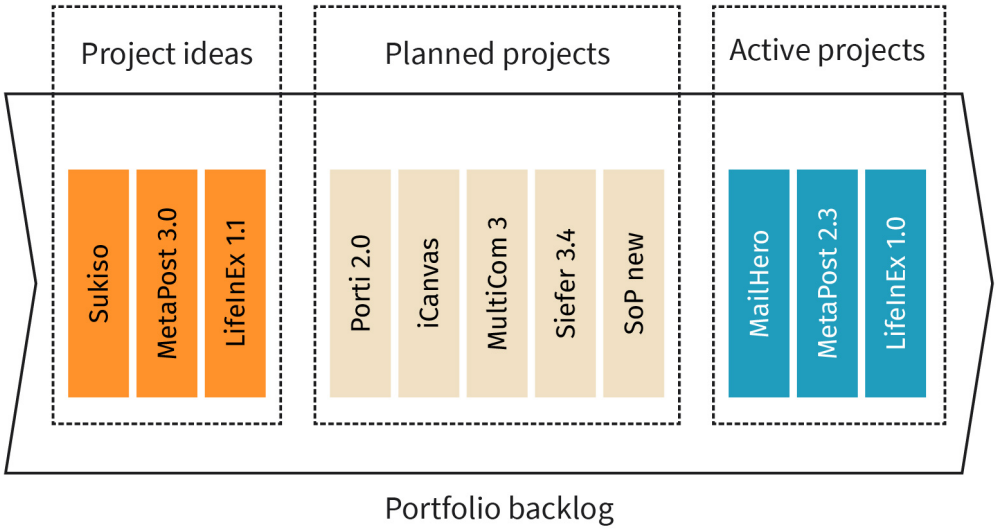
3.2 Agile Portfolio Management

Product development projects are subject to planning at the highest management level and are usually planned based on a product portfolio. Portfolio planning applies to organizations that develop more than one product or application at a time. In practice, projects are typically named after the application that is to be developed or introduced, for example, System Applications and Products in Data Processing (SAP). Therefore, it is not always possible to clearly distinguish between the project and the application or the product when talking about it. This can leave a great deal of room for misunderstanding. For example, one person may be talking about “SAP,” referring to the project of introducing the SAP system, while their counterpart may have the application in mind.

In order to manage the elements of a portfolio according to agile principles, a portfolio backlog is used. Portfolio backlogs contain items such as applications, products or projects, as well as increments and follow-up versions – in the case of continued development of applications or products – like “SAP 1.1” as a further development of “SAP 1.0.” Working with and handling a portfolio backlog is basically the same as working with a product backlog: items at the “very top” or at the “very right” are the most important ones and are the first to be worked on. The portfolio backlog is also subject to a continuous

maintenance process in which items can be added, deleted, re-prioritized or refined. The figure below illustrates a portfolio backlog that groups items into three categories. Project ideas contain items that have been proposed as visions but have not been evaluated yet. Planned projects have been evaluated and are to be implemented. Active projects are projects that are currently being worked on.

Figure 20: Visualizing a Portfolio Backlog



Source: Created on behalf of the IU (2023).

The aim of portfolio planning is to allocate the resources of an organization to specific projects (for example, applications and products) so that they are used in an economically optimal way. Therefore, maintaining the portfolio backlog involves prioritization and re-prioritization of its items as well as deciding whether to implement a certain project. Rubin (2013, p. 270–285) describes several strategies that should be used in agile portfolio planning and groups them into four categories:

1. Scheduling strategies
2. Strategies for including projects into the portfolio
3. Strategies for initiating projects from the portfolio
4. Strategies for continually evaluating active projects

In the following section, the relevance of each category for the portfolio planning process is explained and illustrated using examples.

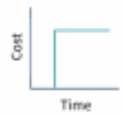
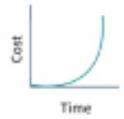
Scheduling Strategies

Scheduling strategies help to prioritize planned projects and put them in an appropriate order in the portfolio backlog. This ensures that resources are allocated to projects in an economically sensible way. Suitable scheduling strategies include the calculation of cost of delay and estimation based on abstract measures.

Consider cost of delay

The delivery date is an important factor when analyzing the profitability of a project or an application. Delays to the completion of an application, a product or a project may carry significant costs. The financial impact caused by delayed completion is known as the cost of delay. Considering the cost of delay is essential when prioritizing applications of projects in a portfolio. Analyzing the cost-of-delay profile is a useful approach for considering the cost of delay when making scheduling decisions. The table below shows examples of typical cost-of-delay profiles.

Table 5: Examples of Cost-of-Delay Profiles

Profile	Name and Description
	Linear cost of delay: The cost of delay increases linearly as a function of time. For example, during the period of delay, external service providers must be involved, which causes a constant increase of cost during that period.
	Immediately increasing cost of delay: The cost of delay is incurred immediately and increases greatly with further delay. For example, there is a risk of a significant loss of market share or revenue if the application is not delivered immediately.
	Fixed Date: One-off costs arise if the planned delivery date is not met. For example, high one-off penalty costs are incurred if a custom programming project is not delivered on time.
	Exponential increase: Over a long period of time there is a very low cost of delay, but then suddenly, the cost greatly increases. For example, neglecting the maintenance of the software architecture suddenly leads to very high efforts in the up-keep and evolution of a software system (technical debt).

Source: Created on behalf of the IU (2023) based on Rubin (2014, p. 313).

Use abstract measures for estimation

When setting up the project schedule, the amount of effort required to carry out a project in terms of working hours is an important factor. In portfolio planning, estimation units and precision levels for portfolio items should be appropriate for the current level of knowledge. Abstract estimation units (such as t-shirt sizes) can be helpful in indicating the estimated cost range at the start of a project, when more specific, detailed measures might lead to a misleading level of accuracy. The figure below presents an example for estimating projects based on t-shirt sizes. Each size represents a concrete cost range expressed in person-days (middle row) or in monetary terms (last row). A meaningful and appropriate division of cost intervals is usually specific to individual organizations.

Table 6: Abstract Estimation Units (Example)

Estimation	Example 1: Cost range, in person-days	Example 2: Cost range, in monetary terms (1,000 EUR: EUR)
XS	1–10	1–15 kEUR
S	10–50	15–30 kEUR
M	50–150	30–100 kEUR
L	150–300	100–150 kEUR
XL	300–600	150–250 kEUR
XXL	> 600	> 250 kEUR

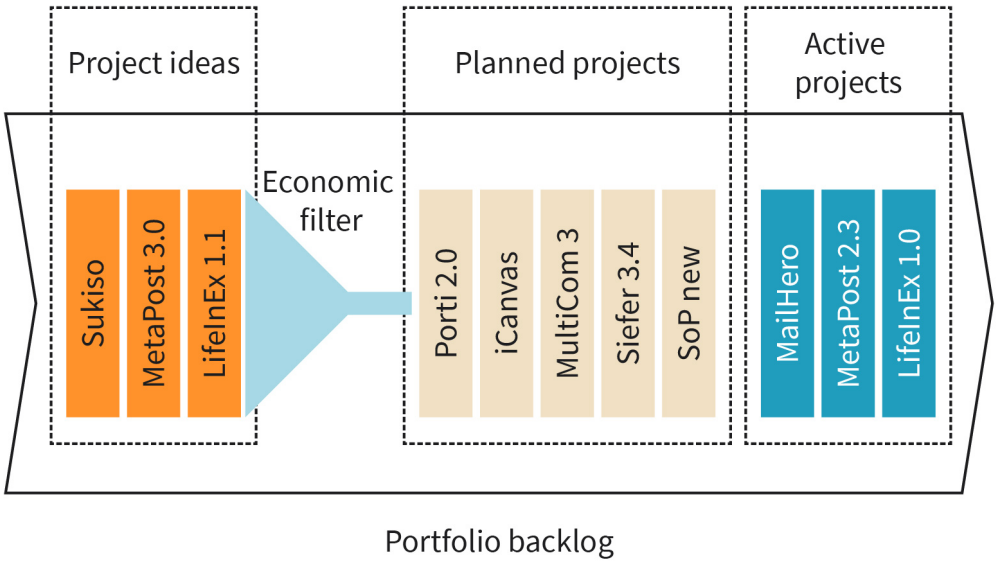
Source: Created on behalf of the IU (2023).

Strategies for Including Projects Into the Portfolio

In order to turn a project idea into a concrete project or to turn a product idea into a product, the stakeholders that are involved in the portfolio planning process have to make a conscious decision about which projects should be included in the portfolio. Considering the profitability of the project or product idea is certainly an important aspect of this decision. The idea is only further considered and worked on if the project or product idea provides an economic benefit. You can think of this as passing through an **"economic filter"** (Rubin, 2013, p. 275) before a project idea turns into a planned project.

Economic filter
The economic filter represents a kind of quality gate in the portfolio backlog that a project idea can only pass if the expected resulting value exceeds the project's cost.

Figure 21: Applying Economic Filters



Source: Created on behalf of the IU (2023).

Moreover, a steady flow of items through the portfolio backlog must be ensured. This means that adding and removing projects from the portfolio backlog should be balanced. Introducing new project ideas into the portfolio in regular, short intervals (for example, on a monthly or quarterly basis) helps with keeping the effort down for evaluating the project ideas and turning them into planned projects. Reducing the project size is also a way to ensure a steady flow through the portfolio backlog. Smaller projects are completed earlier, and thus, leave the portfolio backlog again earlier.

Similarly, planning for frequent and small releases is an effective strategy for getting fast feedback and effectively managing the projects in a portfolio. Large or very large projects or applications bind many resources. If a large project is delayed, all subsequent smaller projects that rely on the resources bound by this large project are also delayed. This also implies that the controllability and reactivity of the portfolio management decreases. If a larger application is broken down and planned in several smaller releases from the beginning, these releases are individually managed in the portfolio backlog and the controllability is more likely to be maintained. In order to support this approach, organizations can specify size limits for portfolio backlog items and aim at breaking down larger projects into several smaller projects. However, switches between projects should not be made too often, because every switch of context comes with an additional adjustment effort for the team members and leads to additional costs.

Strategies for Starting Projects From the Portfolio

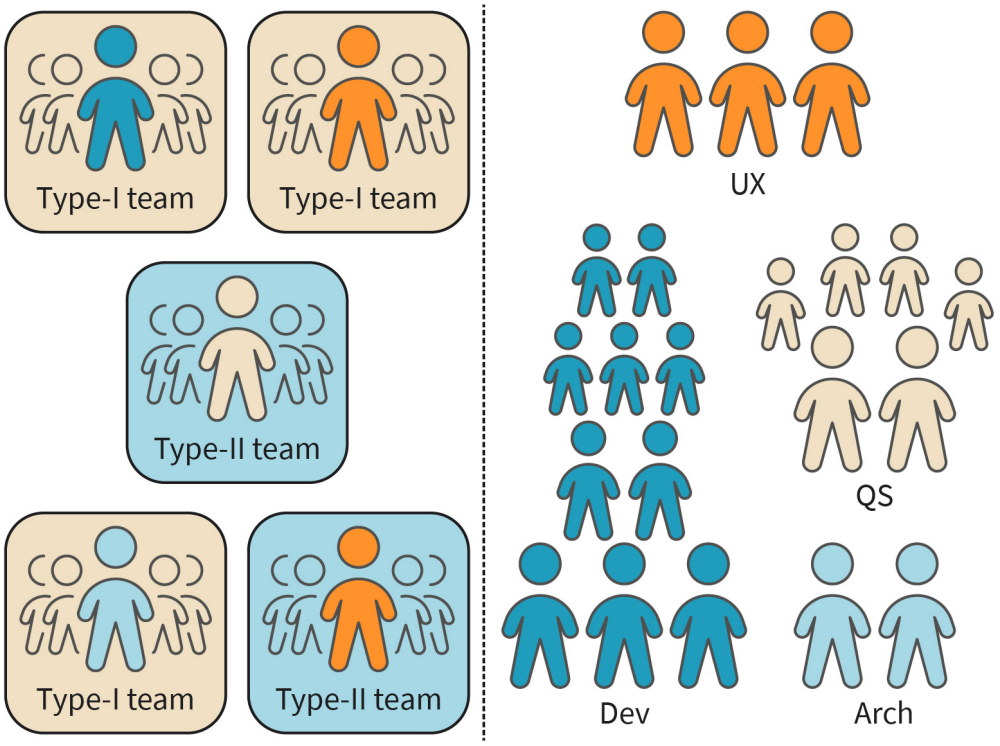
The basic rule for portfolio management is that only as much work should be started as can be efficiently processed. If there are too many active projects in the portfolio, there is the risk that none of them will be processed properly, the employees will feel overloaded, and product quality or adherence to delivery dates will suffer.

The portfolio backlog should contain an adequate number of items in order to prevent idling. Rubin (2013, p. 281–283) suggests defining the work in progress (WIP) limit based on the number of teams rather than on the workload of individual employees. The reason for doing so is depicted in the figure below.

Complete teams that are specialized in certain types of applications cooperate smoothly and are able to produce artifacts that represent complete outcomes. In the figure below, these teams are shown on the left side and their type of specialization is labelled as Type-I or Type-II. In each team, all the competencies that are needed to program (developers), to design the user interface (UX), carry out quality assurance (QA), and design the software architecture (architect) are available.

In total, there are three teams for Type-I applications and two teams for Type-II applications. This results in a WIP limit of five teams, each with a maximum of three active Type-I projects and a maximum of two Type-II projects at the same time. This simple and effective metric makes it much easier to visualize and organize available resources than, for example, on a per-employee basis as shown in the figure below on the right. It can act as an important tool for making decisions about when to start and how to allocate resources to projects.

Figure 22: Teams as WIP Limit



Source: Created for the IU (2023) based on Rubin (2014, pp. 322–324).

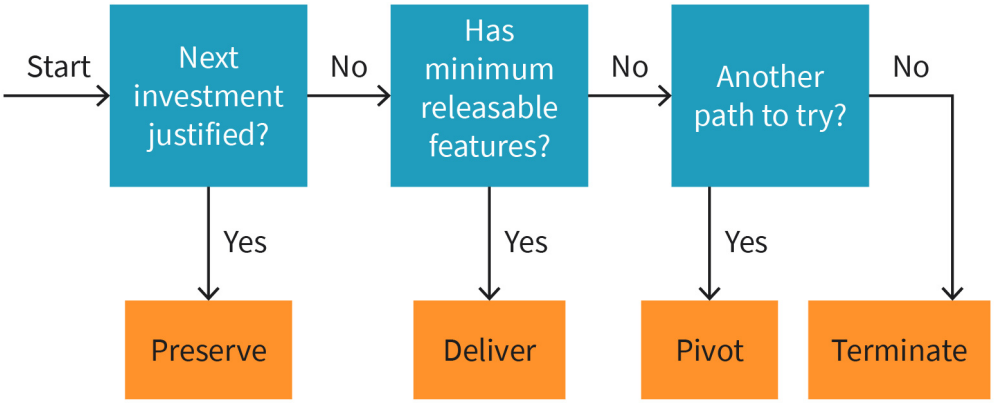
Self-organized teams are of central importance in the context of Scrum. Applying this principle to portfolio management means that an item in the portfolio backlog only moves into active processing when a complete team is available. In this context, the smallest plannable unit of capacity is the team that is able to deliver value to the customer (for example, a first version of the application).

Strategies for Continually Evaluating Active Projects

Typically, a portfolio backlog always contains active projects that are being worked on. As part of portfolio planning, the projects that are currently running must be monitored and an active decision must be made as to whether further resources should be assigned to them. **Marginal economics** can offer support in making these decisions (see figure below).

Marginal economics
In marginal economics, costs and benefits are evaluated from the current perspective (without considering the past).

Figure 23: Applying Marginal Economics to Support Decision-Making



Source: Created by another author based on Rubin (2014, p. 325).

In this approach, continuous checking is required in order to assess whether a further investment in the project can be justified economically. In each portfolio planning session, the current project result, the currently available insights, and the current project constraints are reviewed and used as a basis for making this decision. If necessary, only the currently available product version is delivered, or a completely different solution strategy may even be pursued going forward. Only knowledge-driven development enables marginal economics (as described above) to be applied. This is because knowledge-driven development provides the continuous feedback that is necessary for performing these calculations, which is not the case when applying assumption-based development.

Benefits of Agile Portfolio Management

Agile portfolio planning is made possible by applying the principles and strategies outlined above. As opposed to relatively large portfolio items and relatively rare planning activities, incorporating agility into portfolio planning can specifically minimize delays in reacting to changing constraints and can make the use of resources more efficient. Depending on the company, industry sector and market situation, competitive advantages can be realized by quickly delivering products.

In digital global markets, companies must often expand their market share quickly in order to survive. Therefore, the organization of portfolio management must be designed in a way that responding to change quickly becomes possible. In this way, a first version of the product can be brought to market very early on by combining, for example, monthly reassessments of market opportunities, small portfolio items, frequent releases, and planning based on whole teams. According to the set of agile principles, this first version is then incrementally enhanced and further developed based on the feedback received.

3.3 Organization of Several Teams in one Project

Agile software engineering as described and specified in Scrum can be applied in small projects in a straightforward way. However, in medium or large-scale projects, it is necessary to enable several teams to collaborate effectively on the one project. According to the agile principles, the focus in this scenario should also be directed toward seamless collaboration within and across teams. Approaches to adapting agile software engineering to medium and large-scale projects are usually referred to as "scaling agile." Examples of approaches like this are the Scaled Agile Framework (Scaled Agile, 2021a) and Nexus (Scrum.org, 2021).

Component Teams Versus Feature Teams

If each team is to be able to implement relevant functions for the user or customer, then all roles and competencies needed to do so must be available in each team. For example, when developing a Java web application, the team must be able to design, implement, test, and document the graphical user interface (GUI), the business logic, the database access, and the web service interfaces.

The concrete competencies that are required in a team depend on the project mandate, the organization, and the technologies used. If teams are formed based on system components, different teams can build complete subsystems independently. In contrast, forming teams by roles, for example having a GUI development team, a database development team, and a business logic team, produces cross-team dependencies between the results, and the product functions that are visible to the customers are not created by one team, but across multiple teams.

Large, complex applications often consist of many different components and are connected to many other systems via interfaces. In order to be able to apply agile principles properly in large-scale projects like this, the teams involved must be organized accordingly. In the following section, three organizational forms of agile teams are presented and discussed:

1. Feature teams
2. Component teams
3. A hybrid of feature and component teams

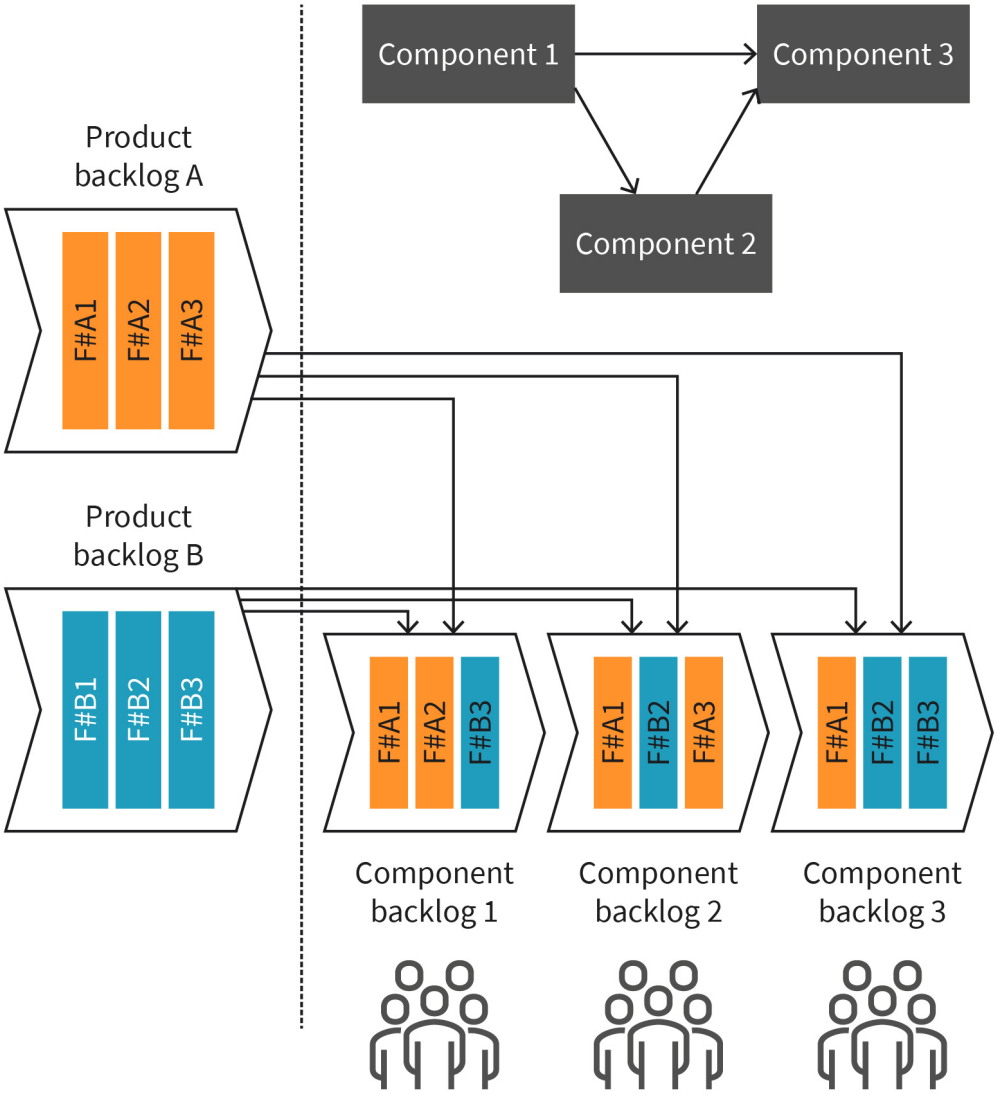
The figure below illustrates a scenario where component teams are working on components of more than one product and are controlled by more than one product backlog. In this example, there are two parallel product backlogs, each of which stands alone. There are three important system components, Component 1, Component 2, and Component 3, each of which is developed, maintained, and evolved by a team dedicated to these components.

The **component teams** are each made up of experts in their field (Larman & Vodde, 2009, p. 156). They each work on the respective system component in order to implement functions from the product backlog. The software architect is responsible for refining product backlog items from Backlog A and Backlog B into corresponding subtasks. These subtasks are then added as items to the component backlogs.

Component team
A component team is formed around the architectural components of a system and consists of experts in their specific field.

The component teams manage their component backlog independently. In this organizational form, there is no guarantee that the priority of the individual product backlogs is mapped to the component backlogs. In the worst case, certain product backlog items are moved backwards again and again because the component team considers other tasks in their own backlog to be more urgent. Organizing teams exclusively in component teams can jeopardize the workflow of a product or a project because there is no team that has a sense of ownership for the whole product across all components.

Figure 24: Organization of Component Teams



Source: Created on behalf of the IU (2023).

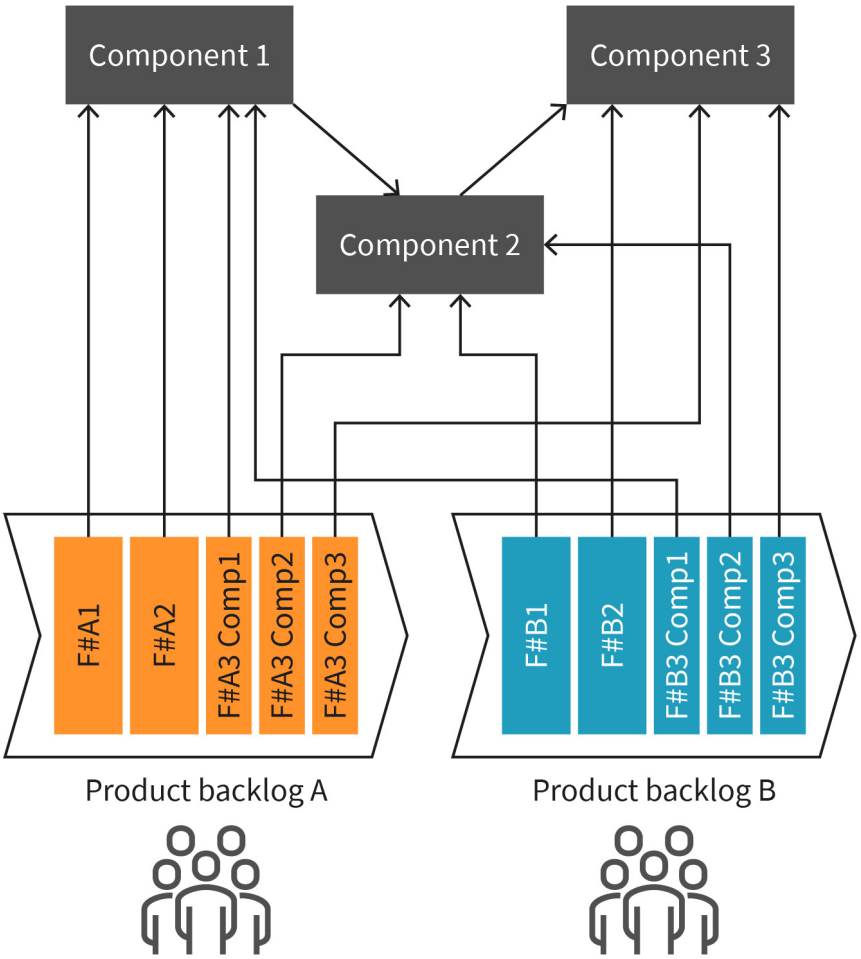
Feature team
This team combines the necessary competencies to do all the work needed to complete a feature.

As opposed to component teams, **feature teams** are not responsible for individual system components. Instead, they are responsible for implementing a specific product feature that could span across several system components (Larman & Vodde, 2009, p. 150). The figure below illustrates an example scenario involving feature teams. There is one team responsible for each of the two parallel product backlogs. If tasks for a backlog item are identified in different system components, separate backlog items are created for each. However, these technical tasks are not written into a component backlog, but remain in the respective product backlog and the team is still responsible for them. In order to do so, the team must have the competencies to work on the individual components. This approach enables an ideal and unhindered processing of the backlog items in the context of each backlog. As opposed to component teams, in this approach the focus is not on the individual components, but on the components' interaction in the respective projects.

Dependencies between the activities of individual teams are now limited to the work on system components. Because there is not one team dedicated to maintaining and evolving the individual components anymore, there is a risk that the independent evolution of the components by different teams will lead to a high level of **technical debt**.

Technical debt
The term technical debt is used as a metaphor to describe consciously or unconsciously built-in quality defects in a software system.

Figure 25: Organization in Feature Teams



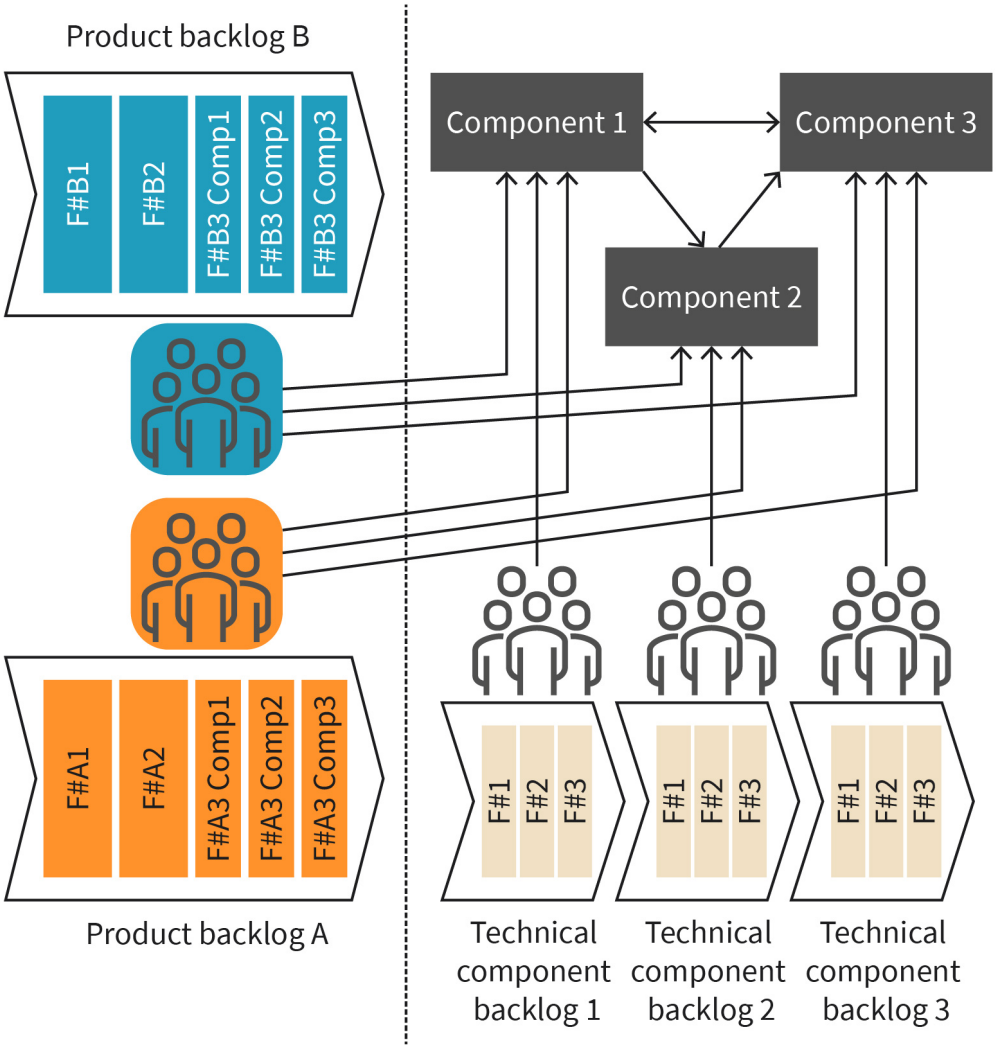
Source: Created on behalf of the IU (2023).

Finally, the organization of agile teams can also be realized in a hybrid form of feature teams and component teams, as presented in the figure below. In this hybrid approach, the processing of product backlog items is organized in feature teams, each of which works independently on the items in all components. By doing so, a smooth product development workflow can be ensured.

Additionally, there are component teams who work on their own component backlog. The component teams are made up of specialists with a very high level of detailed knowledge about the components. The component teams' job is to ensure the integrity of the individual components and to maintain and evolve them in an appropriate manner. Thus, a com-

ponent backlog usually contains very technical items. This hybrid approach ensures the continuous maintenance of the components, even though multiple teams are working on them parallel to one another.

Figure 26: Feature Teams and Technical Component Teams



Source: Created on behalf of the IU (2023).

Furthermore, members of component teams can be specifically assigned to individual feature teams and work there as team members. Thus, component specialists can spread their knowledge on the components into feature teams and pass it on to other colleagues. What is more, by working in feature teams, the specialists gain valuable experience and obtain feedback that they can take back to their component team. They can also take change or adaptation requests and include them in the component backlog. The increased coordination effort that arises due to having several teams working on the same component in parallel could be considered a drawback of this approach.

Application to business practice

In the context of a concrete enterprise or a concrete project, the application of organizational forms for agile teams depends strongly on the size and architecture of the system and the company. On the one hand, teams must be able to build product functions for the customer independently, and frequently change many system components to do so. On the other hand, adequate evolution and maintenance of the components and systems must be ensured. Therefore, the optimal team size in Scrum ranges from between three to nine team members. Additionally, care should be taken to ensure that members of component teams are not required to work on too many feature teams at the same time. For this reason, the alignment of the team organization with the company is generally just as individual as the design of the company-specific software process.

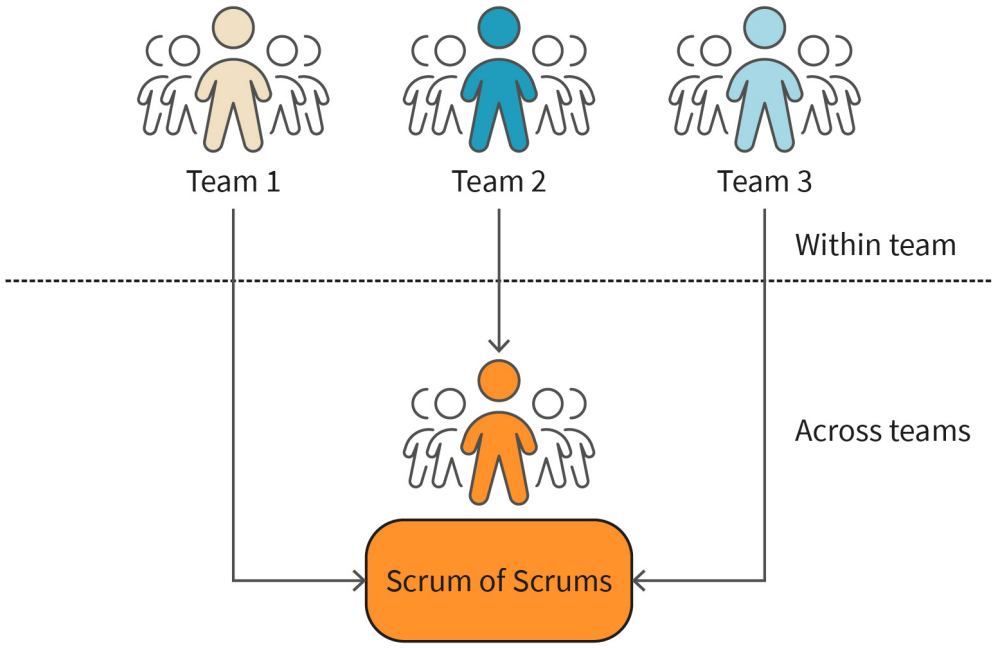
Coordinating and Aligning Several Teams

As previously mentioned, an agile project team cannot be expanded by simply adding more team members to it. From a size of about nine team members upwards, there is a risk that the team will split into smaller groups. If this is inevitable, you should work in several teams from the beginning. The product backlog is an important tool for coordinating several teams. All teams can fill their sprint backlog from the set of product backlog items that have passed the "definition of ready" quality gate. Furthermore, dependencies between the tasks of different teams must be identified and coordinated. Similarly, a flow of information about the current progress as well as about problems and obstacles must be made possible across teams. We will now discuss two techniques that address these challenges:

1. Scrum of scrums
2. Agile release train

The term "scrum of scrums" (Cohn, 2007) refers to an event that is set up for a cross-team update on current progress and problems, similar to the daily stand-up meeting (daily scrum). The figure below illustrates a scrum of scrums, which takes place in addition to all other Scrum events.

Figure 27: Scrum of Scrums



Source: Created on behalf of the IU (2023).

Basically, the scrum of scrums corresponds to the structure of a daily scrum. Representatives from all teams meet in an appropriate, fixed rhythm for a fixed duration and briefly present

- what the team has done since the last event and how it may affect other teams
- what the team will do until the next event, and how it will affect the other teams
- what problems have been solved or what problems still exist that other teams might be able to help with

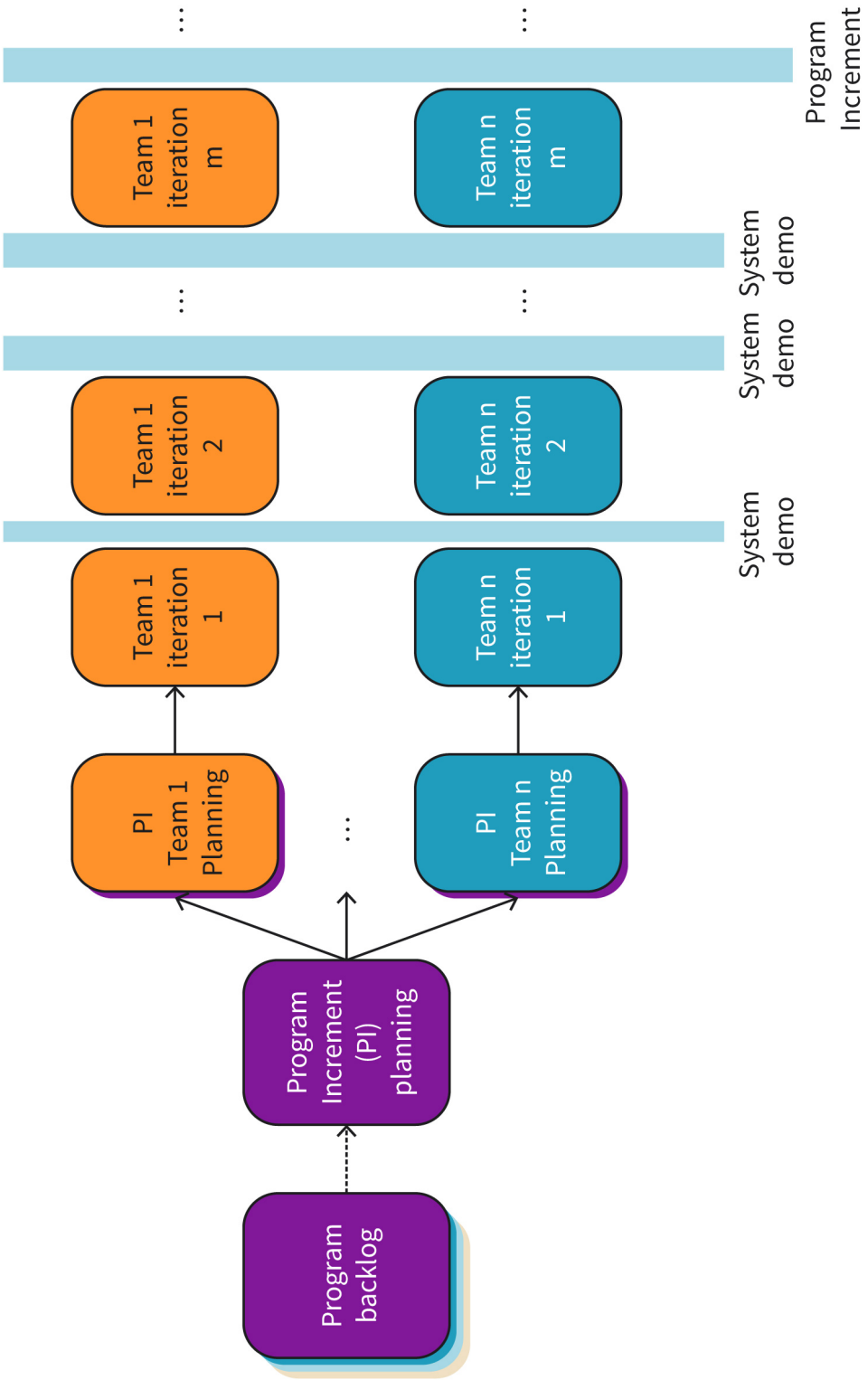
If necessary, a buffer for specific discussions and agreements can be scheduled after the official meeting. Typically, the scrum of scrums does not take place every day, but is often conducted two to three times per week. It is also common practice to choose different team members to represent the team on a rotation, based on their level of knowledge about the topic at hand and their interest in finding a concrete solution. The scrum of scrums can be moderated by a selected scrum master and the chief product owner.

Agile release train
An agile release train is an approach used to synchronize multiple teams by applying a common cadence to development. It aims at aligning the teams based on a shared vision and mission.

The term **"agile release train"** refers to an organizational form in which several agile teams work collaboratively on an application or a product (Scaled Agile, 2021d). It is a kind of "team of teams" that accompanies an application or a product development over several releases. As implied by the train metaphor, the most important element of this approach is the synchronous timing of the train: one team corresponds to a compartment of the train. Each compartment departs from the station at the same scheduled time as the entire train and arrives at the next station together with all other compartments at a scheduled time.

At previously determined dates an executable version of the product or application is delivered. All teams in the release train are organized according to the schedule set by the underlying rhythm (also known as cadence). The figure below illustrates an example scenario for organizing five teams in a release train for one product.

Figure 28: Release Train

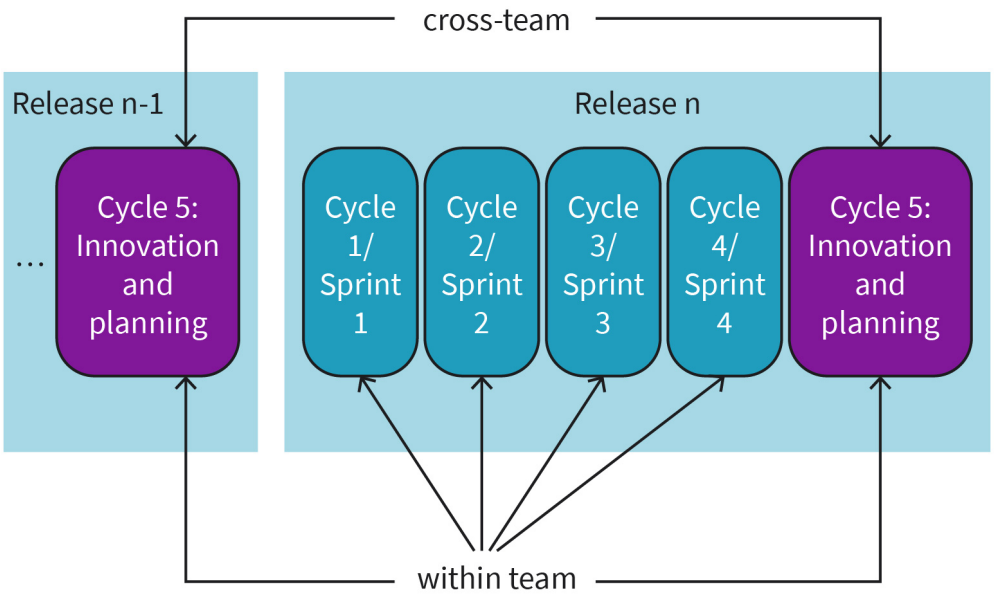


Source: Created on behalf of the IU (2023).

The largest unit of time that is used to do cross-team planning is a release. The structure of a release is shown with respect to time in the figure below. Release planning is conducted in the last cycle of the preceding release in a large cross-team planning workshop involving all members of all teams (Cycle 5 in the figure). The timing of a release cycle is the same for all teams, which means that both the duration of a cycle (also: duration of a sprint) and the number of cycles per release are the same for all teams. The example shown in the figure below includes five cycles per release.

The first four cycles are conducted within the teams, and are aimed at processing the items in the team-specific backlogs. The last cycle is referred to as the innovation and planning cycle (Scaled Agile, 2021b). Concisely formulated, the purpose of this cycle is for the team to "recharge their batteries and sharpen their tools" (Scaled Agile, 2021b). This last cycle serves as a buffer for unfinished work, quality assurance, backlog refinement, refactoring, continuing education, or working on the technical infrastructure. In other words, the innovation and planning cycle provides time for all tasks that were deferred or could not be completed during the focused work on the backlog.

Figure 29: Structure of a Release in a Release Train



Source: Created on behalf of the IU (2023).

For cross-team coordination during the individual cycles, there is the scrum of scrums, whereas cross-team release planning is done in a larger workshop. Scaled Agile (2021c) recommends a two-day event following a proposed standard agenda. The presentation and discussion of the results also takes place in a cross-team setting in the cycle innovation and planning. The structure of the release train basically corresponds to that of a sprint in Scrum. The agile release train implements one of the principles of knowledge-driven development in a cross-team manner: relatively early and frequent feedback loops during development.

3.4 Product and Release Planning

In the next section, we will focus on the role of product and release planning in agile software development, including the planning of backlog items in releases and sprints.

Product Planning

The planning activities at the product or application level include those aspects of planning that are required as guidelines and for controlling the project especially in its initial phase. These include the following

- defining the product vision
- creating the product backlog
- creating the product roadmap

Defining the product vision

The product owner is responsible for defining and continuously developing the product vision (Sutherland & Coplien, 2019, p. 191). The product manager jointly works on this activity with the relevant internal and external stakeholders. Based on the current knowledge and the organizational, technical and domain-specific constraints, the following aspects are documented:

- What is the goal or purpose of the project, that is, what is the project supposed to achieve?
- What is the motivation for the project, that is, what are the reasons for initiating this project?
- A definition of the most important constraints both for the project outcome and the project execution.

In addition to introducing and guiding the concrete project activities, the product vision also serves as an anchor and starting point for workshops and meetings with the stakeholders.

Creating the product backlog

The aim of creating the product backlog is to initialize the product backlog and fill it with the first backlog items. Depending on the situation, requirements may exist already or requirements engineering activities may need to be initiated. Typically, use cases or broadly defined user stories are added as first backlog items.

Creating the product roadmap

The product roadmap is created based on the first version of the product backlog. It defines a minimum set of features to be delivered for each release date based on the planned project duration and the planned release frequency. Rubin (2013, p. 295) characterizes the set of features to be delivered in a release as the “minimum releasable features.” This set should contain the features that are required at a minimum in order to generate

marketable value for the customers. Thus, the release should be planned in a way that it contains the smallest set of absolutely necessary features that represent acceptable value for the customers. When planning large-scale projects that span a longer period of time, a specific release target can be defined for each release. While the product vision is the goal that drives the content-related strategy of a project, the product roadmap shows the timeline for executing the strategy.

If a product is broken down into sub-products or if a large and complex application is divided into individual components that each can be developed in dedicated sub-projects, the planning activities for products or applications described above can be applied to these sub-products or subcomponents as well. However, in this scenario, you must ensure that the release goals of the individual components are continuously aligned with the product release planning as a whole.

Release Planning

Release planning serves two main purposes. Firstly, it helps to coordinate the interface between product development and customers. When a new application release is delivered and deployed at the customer's site, the application often needs to be integrated with other applications in the application landscape using technical interfaces. This step has to be carefully planned and conducted. Secondly, release planning helps to coordinate individual teams amongst themselves with respect to scheduling their work and delivering project results in medium to large-scale projects.

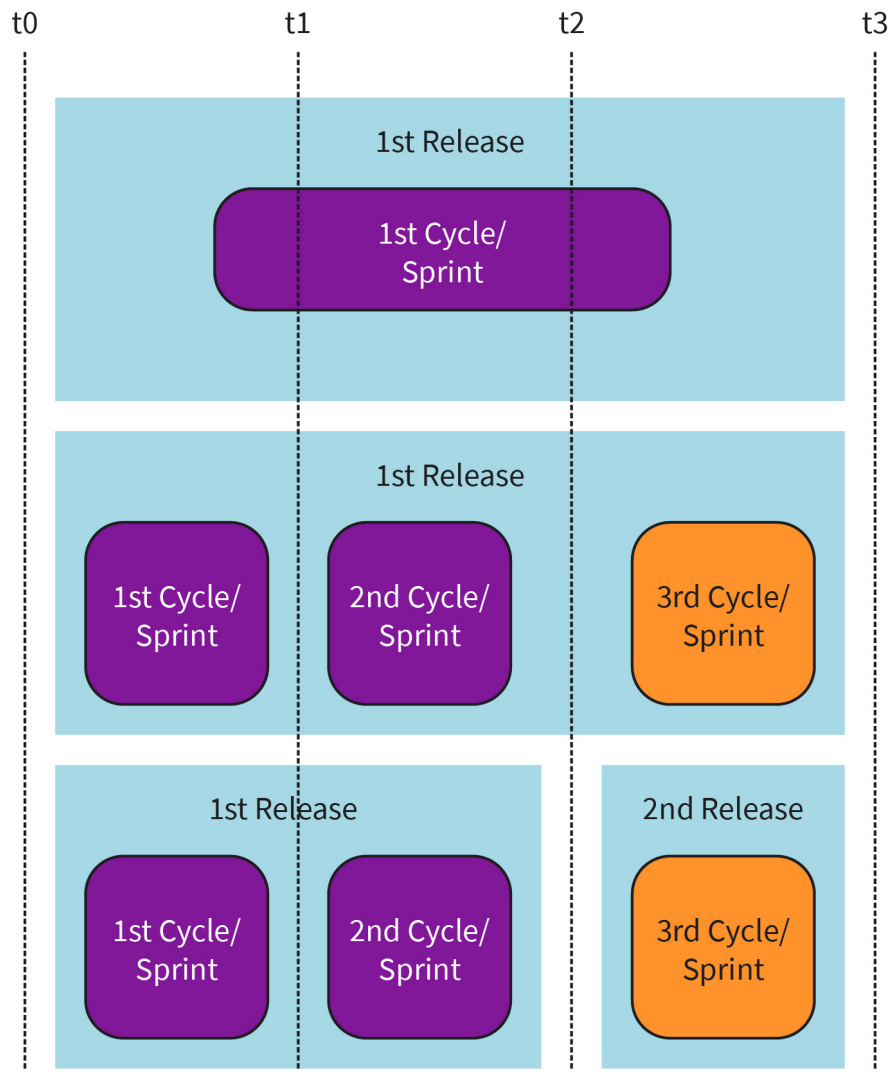
The frequency of individual releases, that is, the release cadence, generally depends on the organization, the type of application, and the size of the project. Relatively small applications that are not tightly coupled to other applications via technical interfaces can usually be delivered more frequently. Large applications with many technical dependencies and many different user groups may delivered less frequently. Applications that are in strategic development and are used to build up a new business area are usually planned differently to mature and stable applications in the area of business support processes. When controlling release cycles, the following parameters must be planned:

- duration of a cycle/sprint: This is the smallest planning unit of a team and period of undisturbed work of individual teams. Typically, a length of two to four weeks is applied here.
- number of cycles per release: This is a parameter that determines how often the results of individual teams are integrated into a deliverable application. In a setting with several teams, this parameter also defines how often the work of the teams is integrated and evaluated. Depending on the application and the project, a release not only includes the delivery of the result currently finished by the project team, but also the deployment at the customer's site.
- number of planned releases: This parameter defines the frequency of delivery of finished results to the customer. Depending on the type of application, frequent releases can be explicitly desired and planned, or only a few releases can be targeted. Large internet platforms, for example, might deliver a new release every week in order to evaluate the acceptance of new functions and continuously enhance and refine these functions based on customer feedback. In contrast, it is typical practice to update central

systems of an organization-specific application landscape that have many dependencies on other systems and that are part of important business processes two to four times per year.

The figure below illustrates three examples of distributing cycles and releases. In principle, all possible combinations are conceivable. The planning parameters presented above control the batch size of a cycle and the feedback frequency. Short sprints of one to two weeks have a smaller batch size than sprints of four weeks and have a higher feedback frequency. Frequent releases require frequent integration, but also provide faster feedback on the actual state of the overall product compared to few releases.

Figure 30: Examples of Cycles/Sprints and Releases



Source: Created on behalf of the IU (2023).

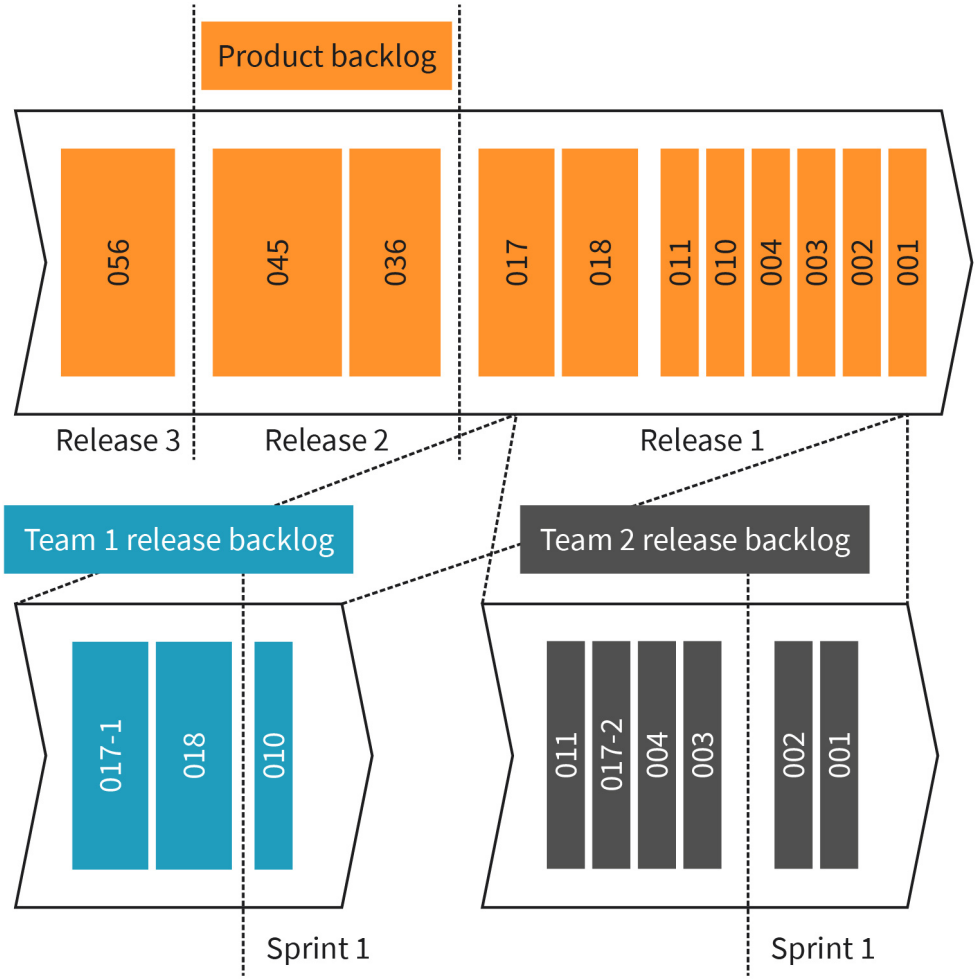
The release cadence is typically influenced by factors such as the process maturity of the company, the market situation, customer requirements, dependencies on external systems, impact on workflows and business processes, and the availability of resources. Even if the development department can provide new releases quickly, IT operations must deploy and integrate them, end users must be trained, changes to workflows and responsibilities have to be implemented in the organization if necessary, and support must be enabled to advise users on working with the changed system functions.

Planning of Product Backlog Items in Releases and Sprints

The figure below illustrates the planning of actual product backlog items for concrete sprints of a release in a medium-size project. In principle, product backlog items are assigned to a certain release. Thus, there is no need to create a new backlog for every release. The items of a release are first assigned to a team, and then, within the team, assigned to a certain sprint of the release.

In the figure below you can see that the current product backlog items are assigned to individual releases. The set of items in a release are called the release backlog. If two teams are working on a project, as shown in this example, the items are assigned to the teams based on the size, velocity, and available skills of the team members. Within a team, work on the items is performed in the planned number of available cycles or sprints that apply to all teams.

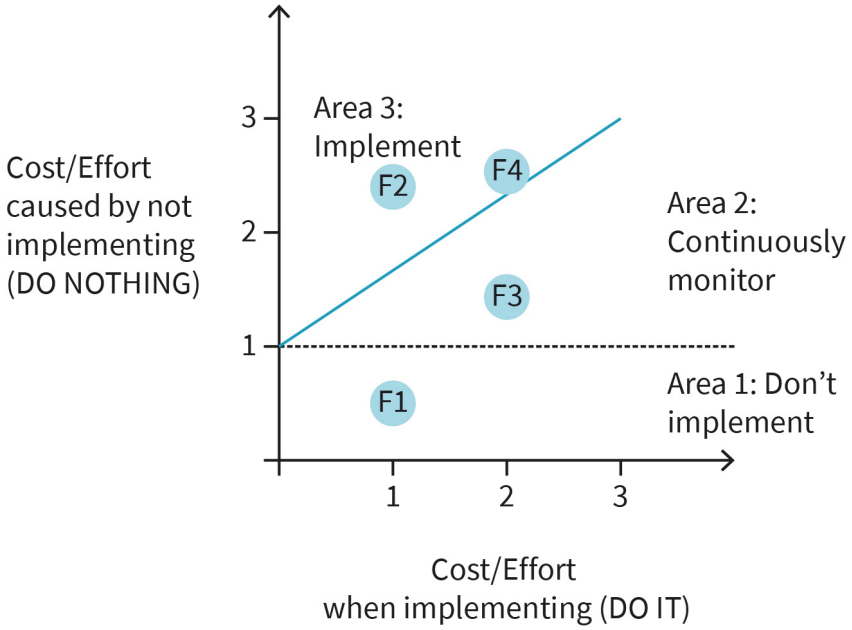
Figure 31: Planning of Backlog Items



Source: Created on behalf of the IU (2023).

The concrete assignment of backlog items to a release or a sprint is based on the principle of avoiding premature decisions, that is, concrete requirements or tasks are only carried out at the last possible moment. The following figure shows an example of how to apply this principle to the planning of product backlog items. For every requirement or activity described by a backlog item, the cost or effort of implementing/conducting it is estimated. The cost that is caused by not implementing the item is estimated in the same way. Both values can be mapped as a point in the coordinate system. The underlying units of estimation depend on the actual context of the project.

Figure 32: Estimating Product Backlog Items

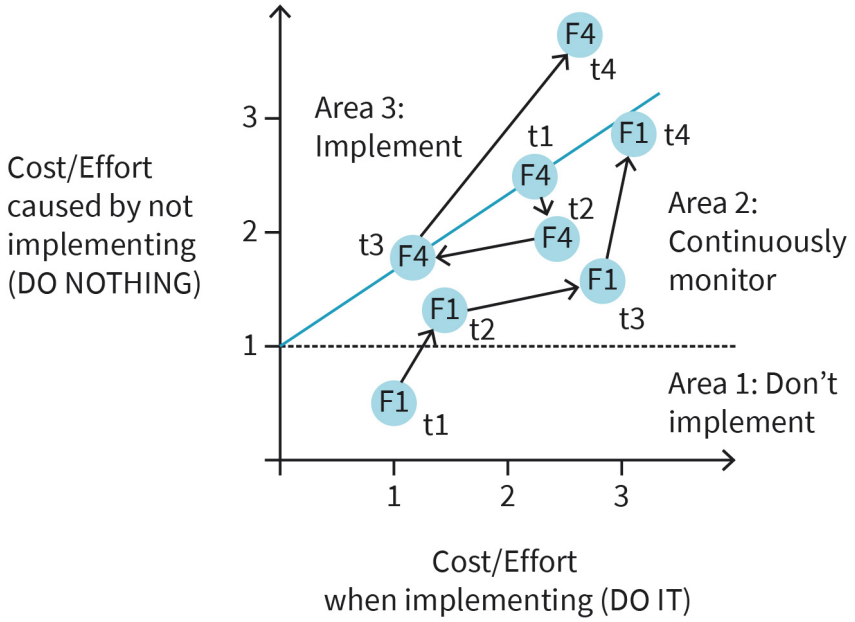


Source: Created on behalf of the IU (2023).

The figure above depicts the following situation: implementing function F1 incurs a cost of 1, not implementing the function incurs a cost of 0.5. Thus, from an economic perspective, it is better not to implement F1. Implementing function F2 also incurs a cost of 1, while not implementing F2 incurs a cost of 2.5. In this case, it is more economically sound to implement F2. The cost for implementing function F3 is estimated at 2, the cost for not implementing F3 at 1.5. Therefore, F3 does not have to be implemented, but should be continuously monitored in the further course of the project. Implementing function F4 is estimated at 2.5, not implementing F4 at 2.7. Here, we have an example of a function that is located at the border between implementation and monitoring. In order to make an informed decision, further assessment criteria should be taken into account. The dotted horizontal line at Y = 1 represents an optional threshold. Items below this threshold are not considered important for the time being.

The figure below illustrates the monitoring of functions F1 and F4 over a period of four measuring points. To measure points t1, t2, t3, and t4, the respective costs were estimated and mapped to the diagram. In this example, you can clearly see that F1 changes from "don't implement" to the border area of "implement." However, F4 moves over different measuring points in the border between "implement" and "continuously monitor." Analyzing the trends also informs the decision about whether or not a requirement should be implemented.

Figure 33: Changing Estimations Across Four Measuring Points



Source: Created on behalf of the IU (2023).

SUMMARY

Truly agile organizations apply agile principles on all planning levels, from portfolio management to release planning. The goal of portfolio planning is to assign a company's limited resources to concrete projects in an economical optimal way. A continuously maintained portfolio backlog is used to manage portfolio items according to agile principles.

In mid to large-size projects, it is important to ensure that several teams can work together effectively. Accordingly, agile teams can be organized in feature teams, component teams, or hybrid forms of feature and component teams, depending on the current requirements.

Techniques such as scrum of scrums, or establishing agile release trains, allow for coordination and alignment between several teams. Planning activities on the level of a product or an application comprise aspects such as the product vision, the creation of a general product backlog, and the development of a product roadmap. The concrete delivery of project results to the customer and their corresponding IT infrastructure is planned and scheduled on the release level.

UNIT 4

AGILE REQUIREMENTS AND IT ARCHITECTURE MANAGEMENT

STUDY GOALS

On completion of this unit, you will be able to ...

- describe the specifics of requirements engineering in agile projects.
- explain and illustrate concepts for the refinement of backlog items.
- apply strategies for breaking down user stories and illustrate architecture-related aspects of user stories using the "Things That Matter" (TTM) matrix.
- define the concept of technical debt and outline methods for managing technical debts.

4. AGILE REQUIREMENTS AND IT ARCHITECTURE MANAGEMENT

Introduction

This unit introduces the specific aspects that need to be considered when carrying out the core software engineering activities of requirements engineering (RE) and IT architecture in the context of agile projects. In the area of RE, you will get to know techniques for breaking down backlog items and tools that can be used to visualize relationships between items. In the area of software architecture management, we will focus on the "Things That Matter" matrix and the concept of technical debt, and approaches to managing these aspects in a structured way.

4.1 Requirements Engineering in Agile Projects

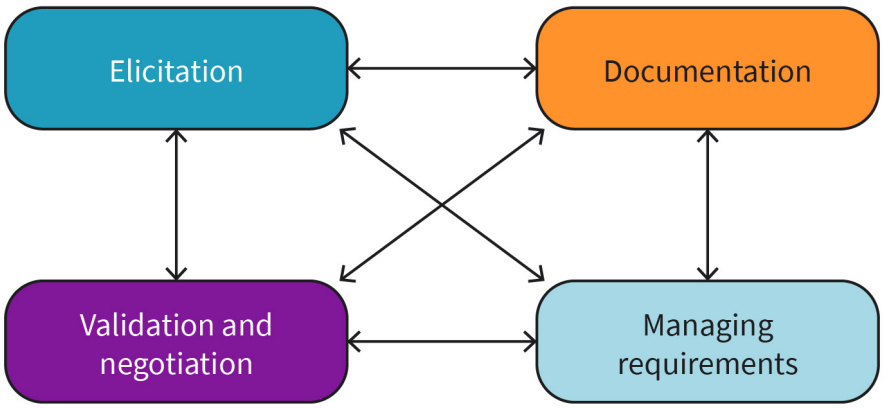
Requirements engineering refers to the software engineering activities of eliciting, documenting, negotiating, and managing requirements. Pohl & Rupp (2015) define RE as:

> a systematic and disciplined approach to the specification and management of requirements with the following goals:
>
> 1. Knowing the relevant requirements, achieving a consensus among the stakeholders about these requirements, documenting them according to given standards, and managing them systematically
> 2. Understanding and documenting the stakeholders' desires and needs, […] specifying and managing requirements to minimize the risk of delivering a system that does not meet the stakeholders' desires and needs (Pohl & Rupp, 2015, p. 4).

Based on this definition, there are four core activities of RE:

1. elicitation of requirements,
2. documentation of requirements,
3. validation and negotiation of requirements, and
4. management of requirements.

Figure 34: Core Activities of Requirements Engineering

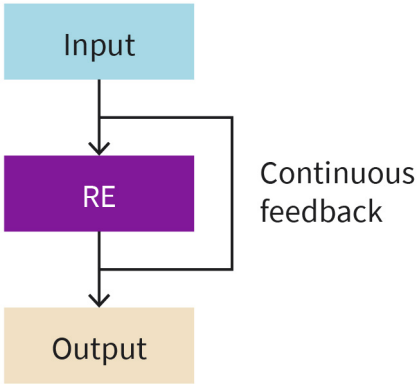


Source: Created on behalf of the IU (2023).

There is no predefined sequence for conducting these activities. In a concrete project, you cannot make a detailed plan upfront, specifying when a certain kind of RE activity is to be conducted using a certain technique. The requirements engineer in charge always decides which type of activity should be carried out next based on the current project situation, the number and availability of stakeholders, and the willingness of stakeholders to cooperate.

There is no fixed workflow that involves first eliciting the whole set of requirements, then documenting, and then negotiating them. Instead, a subset of all requirements is selected, elaborated and refined, documented, validated, and negotiated. Continuously negotiating requirements allows for the realization of the feedback loop as illustrated in the figure below, which is an important prerequisite for the knowledge-driven process of requirements engineering. That is why requirements engineering is fundamentally agile, without the need to change or adapt the classic approach to RE.

Figure 35: Feedback Cycles in Requirements Engineering



Source: Created on behalf of the IU (2023).

However, there are differences in the type of artifacts produced in agile projects as well as in the integration and organization of RE activities in agile software processes. Agile RE does not aim at creating a large requirements document before starting with implementation but focuses on the refinement and elaboration of product backlog items in order to provide the development team with exactly the amount of knowledge they need for implementing the software system. At the same time, you have to ensure a continuous flow. This means that there are as many detailed requirements available as are necessary in the corresponding time frame, without accumulating too many items in your stock of knowledge.

Depending on the project size, individual people, a whole team, or even several teams may work solely on RE and specification. The requirements engineers' capacity depends on the velocity of the teams involved. Throughout the entire project it must be ensured that there are enough items in the product backlog that have passed the definition of ready gate so that there is always enough input for the development teams. It is the chief product owner who is usually responsible for this.

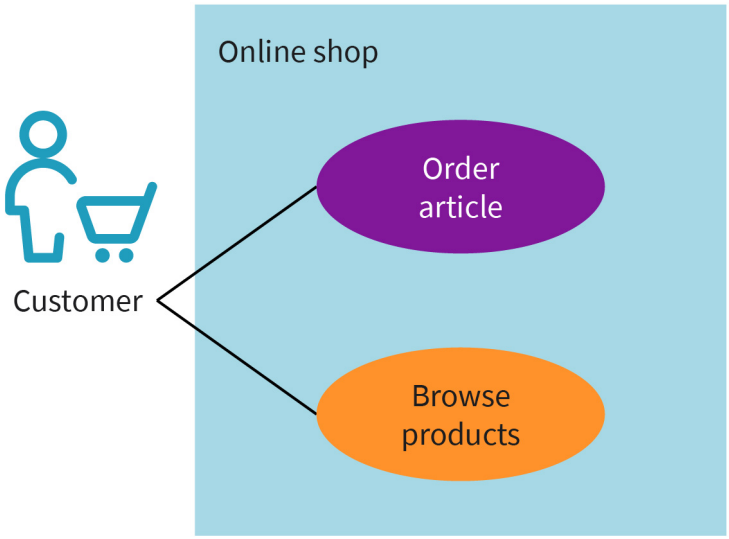
Refinement of Requirements

In principle, there are no methods and techniques for refining and detailing requirements in the product backlog that are only used in agile projects. The main difference between agile and non-agile projects mainly manifests itself in the type of documentation when it comes to RE. In the following section we will introduce refinement techniques that are often referred to in the literature on Scrum or agility. In agile projects, RE activities are often started based on the current version of the product backlog that needs to be refined and detailed considering the constraints imposed by the product vision and the release planning.

Use cases

A use case describes a main function of a software system that is executed to obtain a certain result. It is a very abstract (meaning a very rough) description. Use cases are typically used in early phases of a project in order to identify the relevant functions of a system. They can be seen as placeholders that are continuously refined and detailed in the course of a project.

Figure 36: Visualizing Use Cases in a Use Case diagram



Source: Created on behalf of the IU (2023).

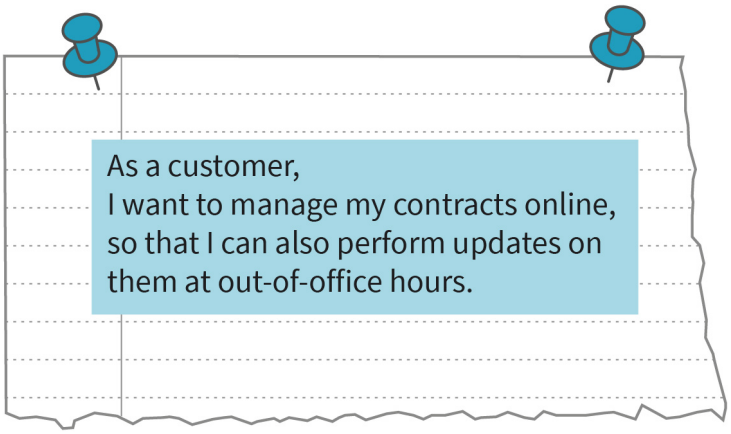
Visually representing use cases, for example, as a **Unified Modeling Language** (UML) use case diagram, is a suitable and simple way of showing the system functions that are offered to its users. Because use case diagrams do not contain any details about the system, they can be used for communicating with the end users or the management.

UML
This is a standardized notation for graphically representing the design of a software system.

User stories

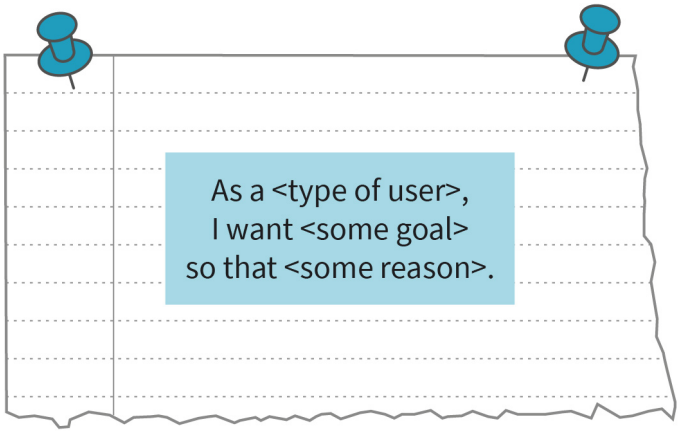
User stories are requirements formulated from the perspective of a certain role. The description of a user story always follows a certain template (Cohn, 2022) as shown below.

Figure 37: Customer



Source: Sandra Rebholz (2022).

Figure 38: Type of User



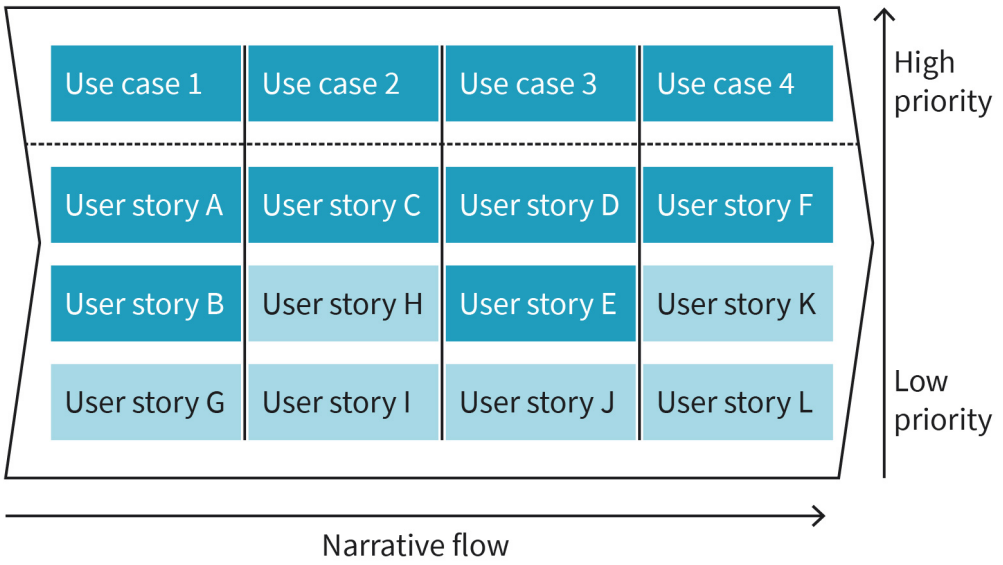
Source: Sandra Rebholz (2022).

It is common practice to write user stories manually on index cards (please see the figure above). The restriction created by the limited amount of space on these index cards encourages stakeholders to formulate user stories precisely. What is more, index cards are easy to sort, group, and duplicates can easily be eliminated. User stories can be formulated on different levels of granularity. A user story can describe both a general use case and a very detailed functional requirement. This makes user stories an ideal fit for the agile approach, which involves continuously refining requirements based on the current state of knowledge.

Story maps

Story maps are a technique for representing dependencies between user stories and between use cases and user stories (Patton, 2014). They provide the big picture of the product to be developed by telling a story of how users would actually use this product. The figure below illustrates how story maps are structured. Use cases or, depending on the project, abstract user stories, denote a main function of the system. All main functions are included in the story map according to the narrative flow, that is, the order in which you would tell the story of the whole product. The headline of the user map contains the main system functions. Below each main function, all user stories are depicted that refine the main function. The order of the user stories also illustrates their priority. In the figure below, user stories are ordered in descending priority from top to bottom. For Use Case 1, User Story A has the highest priority, while User Story G has the lowest priority.

Figure 39: Structure of a Story Map



Source: Sandra Rebholz (2022) based on Brückmann (2015).

It is not usually necessary to finish all user stories of one use case in order to create value for the customer. Typically, there are must-have user stories and user stories that are less important. In the example story map presented above, the product owner highlighted the must-have user stories in dark. These requirements must be implemented so that the release, for example, represents value for the customer.

Patton (2014) compares the layout of a prioritized story map to a backbone and a skeleton. The backbone of the skeleton consists of the main system functions (in the headline of the story map), and the individual user stories are the bones or the ribs that grow out of the backbone. The set of must-have user stories in this skeleton form the "walking skeleton" (Cockburn, 2008, as cited in Patton, 2014). When implementing these user stories, you obtain a minimal working system that provides end-to-end functionality. In our example above, the **walking skeleton** consists of the user stories A, B, C, D, E, and F. Based on the walking skeleton metaphor and the story map visualization, the requirements engineer knows the order in which the requirements need to be refined so that the developers can start implementing the most important functions. Furthermore, the requirements engineer can use the story map as an intuitive illustration of how the set of requirements is structured and use it as an aid for communicating with the stakeholders.

Walking skeleton
This is a working version of a system that provides end-to-end functionality and that implements the minimum set of user stories to provide value to the customer.

Splitting user stories

Larger backlog items must be refined and detailed further as the time of their implementation approaches. Since the main functions of a product are only roughly described at the level of the product vision or the first version of the product backlog, these must be refined and worked out in detail through RE activities. This means that several smaller

user stories replace the large and complex stories and use cases step by step, until you have broken them down into a set of features or tasks that can be worked on by the team within a single sprint.

Lawrence (2020) proposes various patterns that can be applied to split user stories. Based on the following user story, a selection of these patterns is presented and explained by the example:

"As a visitor, I want to buy tickets for the cinema online, so that I can be sure of having a ticket for the desired show."

- **Workflow steps**
  Split a large item by taking a thin slice of the whole workflow first. Every step in the workflow becomes a new user story:
  - Select desired show
  - Select seat
  - Pay
  - Receive ticket

- **Business rule variations**
  In a scenario where different outcomes of business rules are handled differently, split a user story based on business rule variations:
  - Buy a single ticket up to three hours before the show begins
  - Buy multiple tickets up to three hours before the show begins
  - Buy tickets less than three hours before the show begins
- **Interface variations**
  As story can also be split by different user interface options, as follows:
  - Buy on the classical website
  - Buy on the mobile version of the website
  - Buy per iOS app
  - Buy per Android app
- **Major effort**
  A larger user story is split into several smaller stories with similar implementation effort. For example, the first-time implementation of a technical infrastructure can be transferred to similar functions at a much lower effort later on. This story could be split as follows:
  - Payment by VISA credit card
  - Payment by all other credit cards
- **Defer performance**
  A larger story can be split by first making it work and then making it fast:
  - Search for the next available show (slow).
  - Search for the next available show in under five seconds.

This pattern can be applied for any non-functional requirement such as security, scalability or fault-tolerance.

For a full list of all recommended patterns for splitting user stories, students can refer to the Story Splitting Flowchart by Lawrence (2020) and to the Humanizing Work Guide to Splitting User Stories (Lawrence & Green, n.d.). In many cases, there will be several possible ways to split large backlog items into smaller items. The best choice depends on the actual situation in a project. Getting the splitting wrong is not really possible. Later in the project, if you notice that the refinement or estimation of the backlog items becomes difficult, it may be worthwhile to think about adapting the splitting pattern.

4.2 Architecture Management in Agile Projects

In agile projects, the activities in the area of software architecture are carried out in individual cycles. Each development team should be able to implement user-relevant functions within the system independently. In order to do so, the individual development teams must be provided with the necessary knowledge from the preceding work on the product backlog. Depending on the type, size, and context of a project, software architectures can be designed directly and exclusively in the development team.

In projects with a high level of uncertainty about the properties of the final software system (also known as end uncertainty), important specifications and decisions for designing the software architectures can be made by architecture teams. Examples of this type of project include large projects, projects using innovative technologies, or in projects with complex technical interfaces to external systems. However, the difference between knowledge-driven projects and assumption-driven projects is that no attempt is made to make all architectural decisions upfront. Instead, the final architecture is developed iteratively, taking the experiences and the feedback from earlier development cycles into consideration. In order to do so, the development teams must continuously review, validate, and refactor the software architecture. In order to plan for these kinds of activities that do not result in any visible functions for the customer, you can include backlog items of the type “technical work” or “training.” In an agile release train such elements are envisioned in the innovation and planning cycle.

The “Things That Matter” Matrix

The “Things That Matter” (TTM) matrix (King, 2010) is a tool that assigns backlog items to the architectural elements or activities that are relevant for their implementation. The figure below shows a TTM matrix and illustrates how it is used. Each row refers to one backlog item which is identified by the first column. The remaining columns contain aspects that are relevant for the architecture and may influence the estimation, prioritization, and the work organization within the team. There is no fixed specification for these columns. The actual layout of the TTM matrix is dependent on the type of project and the way that the project team works. In the figure below typical layers of an enterprise web application were selected as columns. Each cell of the table shows the complexity, or the expected implementation effort roughly estimated in t-shirt sizes. If the given architectural element

is not relevant for the user story, the corresponding cell is left empty. The higher the weight of a cell – indicated by the t-shirt size – the more likely it is to affect the design of the software architecture.

Table 7: Example of a TTM Matrix

Backlog item/User story	GUI	Business logic	Web service	Database connection	Enterprise service bus
A	XL	S		L	
B		L	S		
C		M			M
D		S	L	L	
E	XL	M			
F	S	S			
G		L	M		M
H	M			M	

Source: Created on behalf of the IU (2023).

From the figure shown above, you can, for example, derive the following statements:

- **User Story A** involves a very complex or elaborate graphical user interface (GUI) and a complex database connection, but only a relatively simple business logic.
- **User Story B** does not have a GUI, but a relatively simple web service interface and a quite complex business logic.

The TTM matrix is created and maintained by software architects or the development team. It is used as a tool for estimating individual backlog items as well as for planning tasks within a team or between several teams. Moreover, the TTM matrix provides important information on which items are relevant for the software architecture and to what extent, and which items are required to be designed and implemented with specific care.

Technical Debt

The term “technical debt” is used as a metaphor to describe quality defects that are consciously or unconsciously built in to a software system (Cunningham, 2009). If a requested function is implemented very quickly, but no effort was put into creating good software architecture and into carrying out appropriate testing, this enables fast delivery. However, the software system now contains technical debt that leads to higher implementation efforts for all further changes to the system when compared to a system without technical debt.

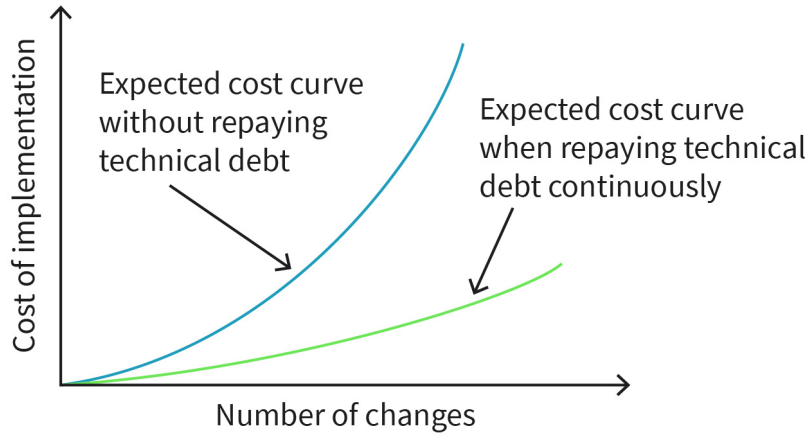
In this course book, the term technical debt is used according to the definition of Röpstorff & Wiechmann (2012, p. 328): “Technical debts are the consequences resulting from developing software too fast, from insufficient software quality, and from an insufficient software architecture.” Thus, avoiding technical debt and shortening time-to-market are two competing goals.

The metaphor was originally conceived to describe this type of effect in simple terms so that anyone lacking the appropriate IT background could also understand it. In the same way as with financial debts, taking on technical debts also incurs interest, in this case in the form of higher implementation efforts needed for future requirements or the additional effort needed to reduce the technical debts.

The cost of repaying technical debt later is certainly higher than the effort required to implement a solution without debt from the beginning. For example, when the focus is placed on fast delivery in product development from the start, more and more technical debt will be accumulated over time, which results in longer implementation times for future changes.

There is also the risk of debt overload, that is, the point when a system is only repaired in order to keep it up, but by doing so all resources that were actually planned for evolving the system are used up. From this perspective, a backlog that incurrs technical debt involves a higher entrepreneurial risk than a solely investment or modernization backlog, where there is usually no interest to consider. The figure below illustrates the relationship between technical debt and cost of change using two cost curves.

Figure 40: Relationship Between Technical Debt and Cost of Change



Source: Created on behalf of the IU (2023).

If the project team continuously documents the identified technical debt and appropriately illustrates this to the product owner or the those responsible for resource management, the repayment of debt can be planned and decided for by means of technical work such as refactoring. The actual planning of these tasks is done via backlog items of the type “technical work.”

However, compared to financial debt, technical debt cannot be measured or quantified precisely (Fowler, 2019). As illustrated by the figure above, this means that the increase of the green curve cannot be predicted reliably and becomes less predictable as the life cycle progresses without any technical work being carried out. At some point, the development team is no longer able to estimate the effort required to do further development on the application.

Nevertheless, the development team can make a qualitative assessment of the state of the system by continuously reviewing and documenting technical debt. Thus, they can estimate the required efforts for repayment. Rubin (2013) describes the following risks due to technical debt:

- unpredictable tipping point: At a certain but unpredictable point in time, seemingly small additional debt has a high impact on actual technical debt. With every quick change to the system, more debt incurs than value is created.
- rising development times: The actual time to implement a requirement increases, which means that velocity decreases as debt increases.
- rising number of technical defects: The higher the debt, the less structured the software architecture and the higher the risk for accidentally introduced defects, because structured development is not possible. In combination with insufficient quality assurance, the risk of delivering unknown defects increases.
- increasing estimation uncertainty: The more debt is accumulated, the less predictable the costs, dates, and delivered functions are.
- increased frustration: The developers become unhappy because they cannot perform as expected, management is frustrated because agreed costs and dates cannot be met, and users are frustrated because they have to work with a system full of defects where improvements or new functions are only implemented very slowly or not at all. (2013)

Managing technical debt

The following activities address the targeted and structured management of technical debt (Rubin, 2013):

- managing the accrual of technical debt
- communicating and visualizing technical debt
- repaying (reducing) technical debt

The accrual of technical debt can be managed by introducing appropriate development processes and by applying appropriate technical infrastructures for automated testing and continuous integration. Using a strong definition of done is an important organizational measure to minimize technical debt during the development process. If the definition of done contains appropriate items related to quality assurance that are mandatory and checked strictly, the increase of technical debt (for example, by leaving out tests or by not updating technical documentation) is prevented.

Another measure to prevent the accrual of technical debt is scheduling dedicated refactoring times, in which the team routinely carries out technical work on the system. In the agile release train, there are regular cycles for carrying out technical cleaning and repair work. Technical practices that reduce the accrual of technical debt include automated software tests and continuous integration approaches (Duvall et. Al., 2007).

If unintended technical debt is discovered or the product owner or management has decided to deliberately take on technical debt, it must be appropriately documented and visualized. On the technical level, communicating and visualizing technical debt can be done based on the product backlog. Technical debt can be managed as a specific item type in the product backlog and when it is identified, the team can immediately add it as a new item. Thereby, these tasks become part of the regular maintenance process of the backlog items, they are represented transparently, and can be scheduled by the team and product owner just like any other activities.

On the economic level, technical debt can be represented as the sum of the estimated effort in monetary units for these backlog items, for example 300,000 EUR. Alternatively, the decreasing velocity can be documented and converted into monetary terms. If the average velocity of a team with fixed costs of 30,000 EUR decreases from 30 story points (1,000 EUR/point) to 20 story points (1,500 EUR/point), the cost per story point increases by 500 EUR. A scheduled amount of 150 story points still open in the product backlog would then lead to an extra cost of 75,000 EUR for the development project or an extension of the development time by exactly 2.5 months.

Generally speaking, technical debt does not need to be avoided at all costs. Depending on the project, purpose, and stage of the system within its scheduled life cycle , technical debt can be taken on deliberately. This debt may be repaid only partly or not at all. Usually, throw-away prototypes, short-lived products, or systems that have reached the end of their life cycle are no longer maintained.

Repaying technical debt should be organized in the same way as implementing new requirements. This means that technical debt is repaid step by step in economically appropriate batch sizes. Here, technical debt causing the highest interest and items with the largest negative impact on velocity should be serviced first. By managing and scheduling technical work items in the product backlog, you can organize the continuous repayment of technical debt. If, in addition, the team develops a culture of accountability for dealing with identified technical debt, which aims at quickly fixing things instead of looking for somebody to blame, many small debts and quality defects can even be fixed by immediately refactoring work after their identification.

SUMMARY

RE in itself implements agile principles, because its core activities are executed continuously and iteratively during the course of the project. These activities include selecting individual requirements, eliciting information for detailing them, documenting, reviewing, and negotiations. In

this way, a feedback loop is realized: an important prerequisite for knowledge-based management. The goal of RE in agile projects is the refinement of backlog items. It must be ensured that the development team has a sufficient understanding of the backlog items that are selected for implementation in the current development cycle.

Requirements in agile projects are often represented as user stories that describe the requirement from the perspective of a certain user's role. In addition, story maps are a suitable tool for visualizing the dependencies between various user stories of a software product.

There are different splitting strategies for refining backlog items, for example, splitting by workflow or by business rules. The continuous review, assessment, and revision of the software architecture by the development teams is included in the backlog through elements of type technical work or training.

An important goal of activities related to software architecture is the targeted management and elimination of consciously or unconsciously built-in quality defects, known as technical debt.

UNIT 5

AGILE TESTING

STUDY GOALS

On completion of this unit, you will be able to ...

- distinguish agile testing from classic testing.
- describe the organizational prerequisites that are required for agile testing.
- choose and apply appropriate testing methods and tools on different testing levels.

5. AGILE TESTING

Introduction

This unit provides an overview of the application of agile principles in various software testing activities. After summarizing the core principles of testing, the differences between agile and classic testing are highlighted. In addition, the organizational requirements for quality assurance that agile testing entails are discussed. Finally, agile testing methods and tools are explained in detail and discussed in the context of each testing level.

5.1 Basics Principles and Requirements for Quality Assurance Organization

Certain organizational preparations are required in order to apply agile principles to testing activities in the development process. To understand why this is necessary, we first need to clarify the approach of agile testing and how it differs from traditional testing approaches.

Classic Testing

Testing level
In order to handle testing complexity, tests are categorized into different testing levels and are executed in different testing environments.

In software engineering, the term "testing" refers to all dynamic quality assurance procedures. Testing software means bringing a system or parts of a system to execution, providing concrete input data and evaluating the system behavior or the generated result. Thus, tests are diagnostic measures for checking and evaluating the quality of software artifacts after their creation. Because large software systems are built from many different components and also have a large number of interfaces with other software systems, testing software is a complex task. For this reason, a software system is not fully constructed first and then tested as a whole. Instead, it is tested on individual **testing levels** and in different testing environments (in classic approaches, as well as in agile testing). On the first level, individual components are tested. These tests are called unit tests and are typically implemented by the software developers. They are executed in the same environment in which they are developed. The interaction between components is tested in integration tests. For these tests, a special infrastructure is set up in the form of an integration environment that enables execution of the software components being tested as well as providing production-like test data.

Both testing levels described above are **open-box tests** (also known as white-box tests). This means that the program code is known when creating and executing the test cases. Consequently, the code structure is used as a basis for creating the test cases and for fulfilling, for example, certain test coverage criteria. If these white box tests have been passed successfully, the next step is to carry out closed-box testing which tests the software as a functional unit. In closed-box testing, the internal structure (code structure or component architecture) is irrelevant.

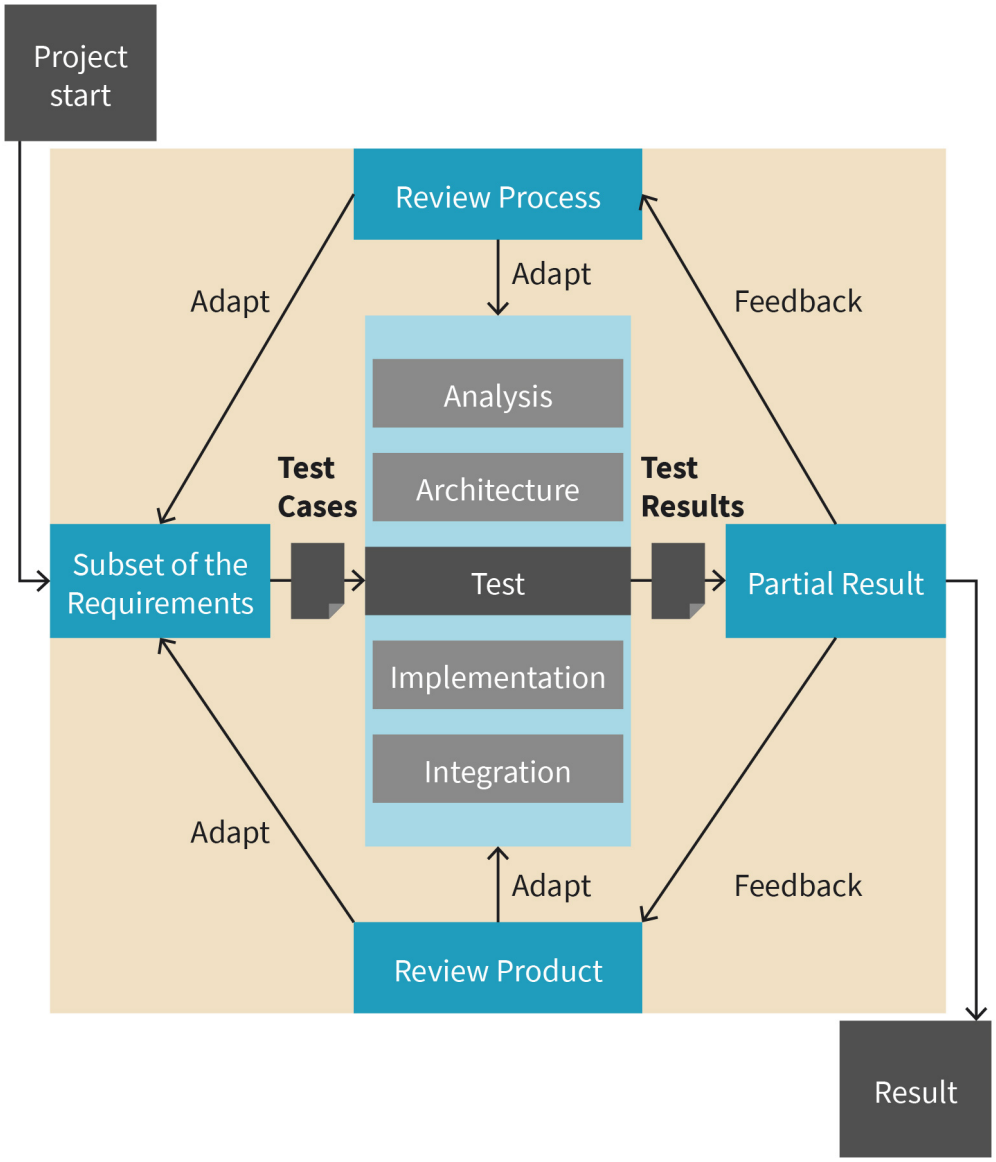
These tests are part of the system testing level. System testing includes both tests of the functional requirements and the quality requirements such as the behavior of the software system under load conditions (load tests). System tests are executed in a special, production-like testing environment. Following the completion of system tests, acceptance tests are carried out. These tests are often done by end users and focus on the functional appropriateness of the software system embedded in the business process. It is common practice to perform these tests in the actual production environment.

Open-box and closed-box tests
Depending on whether the source code of the software is taken into account during testing, the tests are called open-box (source code is considered) or closed-box tests (source code is not considered; also known as black-box tests).

Differences Between Classic and Agile Testing

There is no common definition of agility in software engineering. The same applies to agile testing. For the purpose of our discussion here, the characteristics of agile testing will be described based on the generally accepted agile principles. We will also address the question of how testing is considered in these agile principles and in common software process models.

Figure 41: Agile Testing in Knowledge-Driven Software Engineering



Source: Created by another author based on Rubin (2015, p. 70).

One important difference between agile and classic software development is the basis on which important decisions are made in a software project. As shown in the image above, agile software development aims at making decisions based on the insights gained throughout the process,not on assumptions. This applies across all software development activities. Delivering these **early insights** without any delay is the major goal of agile testing.

Early insights
Agile testing aims at delivering early insights and feeding them back into the software development process.

The characteristics of agile testing stem from this goal. Accordingly, agile testing means the following:

- test as early and as often as necessary
- automate as many tests as possible (except for exploratory testing carried out by experts)
- focus on basic functionality first and then apply other types of testing as necessary ("first make it run, then make it fast")
- work in an interdisciplinary manner, this also means including the developers, depending on the type of test

Early Testing

Agile software process model frameworks such as Scrum do not explicitly consider testing. This might seem strange at first glance, but it is a logical implication of the agile principle of fast feedback. In contrast with the classic approaches of software development, software is no longer designed on a large scale and in long-lasting design phases, then implemented and tested (as it is in assumption-driven software development). Instead, smaller, easy-to-handle functions are implemented, which are consequently tested in a much faster cycle.

Test Automation

Although quick release cycles offer many advantages, one major disadvantage must be considered: **regressions**. Regressions are errors that are introduced by source code modifications to software components that have already been tested. This means that changes to the program can cause errors in already existing parts of the program. In order to detect these errors, all tests must actually be repeated for each modification. Obviously, if release cycles last only a few weeks, the effort involved is far too great to be done manually anymore. Therefore, automation on all test levels is an essential prerequisite for agile testing. However, it is important to note that automation alone is not a unique characteristic of agile testing. Likewise, there are many companies organized according to classic development approaches that use test automation. The difference is that automation is optional in classic testing, but it is mandatory in agile testing.

Regression
Defects in the source code that are introduced or uncovered by code changes in other areas of the software.

Testing the basic functionality

Another difference between classic and agile testing is the sequence of the testing levels. While processing all testing levels strictly in the given sequence is the typical approach in classic testing (especially in "enterprise software engineering"), this dogmatic approach is softened in agile testing. Although it is clear that unit tests are carried out early while manual acceptance tests are carried out late, part of the manual tests or the performance tests may be carried out using early prototypes. This flexibility is important in order to get a feel for the performance or the integrability of the software into the production system at an early stage, rather than having to wait until the deployment of the software.

Irrespective of this flexibility, the basic principle of "first make it run, then make it fast" should be adhered to, which means that the functionality should be tested first, followed by the non-functional requirements. This safeguards against spending too much time on optimizing functions in terms of performance and usability, which may mean that important functionalities are not implemented due to lack of time or budget.

Testing in interdisciplinary teams

Due to the short development cycles, a clear organizational separation between the roles of architect, developer, and tester is not necessary in agile testing. Instead, the whole team is responsible for developing working software, and thus, also for testing. Nevertheless, in practice, there is always a certain need for people who are experts in their respective roles. According to the classic concept of the division of labor, the **testers** and developers would be placed in different departments. However, in agile projects, a strict separation of responsibilities within the teams is not desired. Sometimes, the tester will also take on the role of a developer, and conversely, the developer will be responsible for testing their code before integrating it with other components. The table below summarizes the differences between classic and agile testing. Note that both approaches are described in their extreme forms. In practice, hybrid forms are typically found both in classic and in agile testing.

Tester
In agile testing, there is no strict separation between tester and developer. It is the responsibility of the developers to test their code (at least on the level of unit tests).

Table 8: Differences Between Classic and Agile Testing

	Classic testing	Agile testing
Timing	Based on pre-determined testing phases	As early as possible and as often as necessary
Separation of roles	Often clear separation between developers and testers; no overlaps allowed in order to maintain the objectivity of tests	Interdisciplinary team of developers and testers; overlaps desired in order to improve communication
Organizational structure	Fixed, organizational structures separate testers from developers (often also spatially); different departments for developers and testers.	Testers and developers work together closely (also spatially) in one team
Management	Test managers plan and monitor tests.	Team is self-organizing, and thus also organizes the testing
Responsibility for testing	Tester	Team
Automation	Optional	Mandatory
Testing levels	Sequential: Unit Test -> Integration Test -> System Test -> Acceptance Test	Testing levels are applied as needed; functional testing before quality testing

Source: Brückmann (2015).

Organizational Requirements of Agile Testing

In order for an IT organization to apply agile testing, a suitable tool infrastructure must be available and organizational preparations must be made in advance. This means that it is not possible to convert to an agile approach mid-project. The following preparations are necessary for agile testing on all test levels:

- shifting the testing effort (less manual, more automated unit tests)

- shifting testing tasks to the developers
- disciplined work organization of testers
- well thought-out test data management
- close cooperation between development and operation
- involving end-users in testing

Let us look at a first example of such changes: shifting the testing effort from higher testing levels to lower testing levels. Because errors become more expensive the later they are discovered in the software development process, it is a common goal to achieve a high test coverage on the early level of unit testing (known as the principle of **failing fast**). Therefore, there are significantly more unit tests than there are system or acceptance tests. Shifting the proportion of test cases from higher testing levels to the lower testing levels also affects the work of the developers. While in classic testing the test cases are still written by the testing department, in agile testing, the majority of test cases are written by developers. What is more, agile testing places higher demands on test coverage. Therefore, developers invest much more time in creating test cases than with the classic testing approach.

Fail fast principle
The principle of failing fast is based on the knowledge-driven nature of software engineering and refers to the goal of making errors visible early in software development.

The approach of test-driven development (TDD) allows developers to integrate the writing of unit tests directly into the programming process instead of putting the main effort into testing after coding. According to TDD, the developer works in a cycle of three essential activities. First, a test for the new functionality is written. Then, the program code is written so that the test can be passed successfully. In the second step, the focus is on passing the test, but not on writing high-quality code. Therefore, there is a third step of refactoring, in which the code is cleaned up and improved. After that, the next cycle starts by specifying a test for the next piece of functionality.

In each cycle, the scope of the newly added functionality should be small enough so that one cycle only lasts a few minutes. The degree of agility can be varied based on the cycle length or the amount of functionality that is implemented per cycle respectively. Therefore, less agile interpretations of the TDD approach typically choose a longer cycle length.

Obviously, introducing agile testing methods also has an impact on the role of the software tester. Test automation does not imply that testers will have less work. Because there is no test manager in the agile approach, testers must be self-organized. In the face of small release cycles that leave very little time for preparing and carrying out the testing, this can be a hard job. This is in stark contrast to the classic approach, where quality assurance (assuming the project is not behind schedule) is given several weeks to review specifications and create sophisticated test scripts.

The management of test data also requires special consideration in agile testing. Because test cases are naturally related to test data, applying agile testing means providing this data in an agile way. Consequently, test data should always be available and should be created automatically where appropriate. Agile testing requires a structured test data management strategy based on solid methodical foundations. This is to ensure that the test data always fits the tested software. In the case of changes to the software, the test data must also be checked and updated if necessary. If the same test data is used by other test cases, the test data must be managed in a version management system, just as the

source code is managed. In doing so, the relationship between the test data, the test cases, and the version of the program must be considered and documented. These tasks require close collaboration between testers and developers.

If agile testing is implemented both by using test automation and by the automatic provisioning of the test infrastructure, this transformation has implications for the entire IT organization. In an agile context, the traditional separation of development and IT operations is a disadvantage, because developers no longer deliver software changes every few weeks, but almost daily, which in turn have to be deployed to the target environment by members of the IT operations department. Because of this increased demand for cooperation between development and operations, companies that are strictly organized according to agile principles often introduce the **DevOps** approach. The DevOps approach involves measures that tightly integrate development and operations. For this purpose, members from IT operations and developers work together in one interdisciplinary team instead of in separate teams. They then work jointly on implementing technical requirements beyond department boundaries.

DevOps
The DevOps approach denotes the combination of a specific mindset, practices, and tools that aim at integrating and automating the processes between development and IT operations.

Applying this approach prevents conflicts that are difficult to understand from a domain perspective. For example, if the developers and members of the IT operations department work separately, it might only become apparent during deployment to the testing environment that the software architecture of a new component cannot be mapped to the existing IT infrastructure.

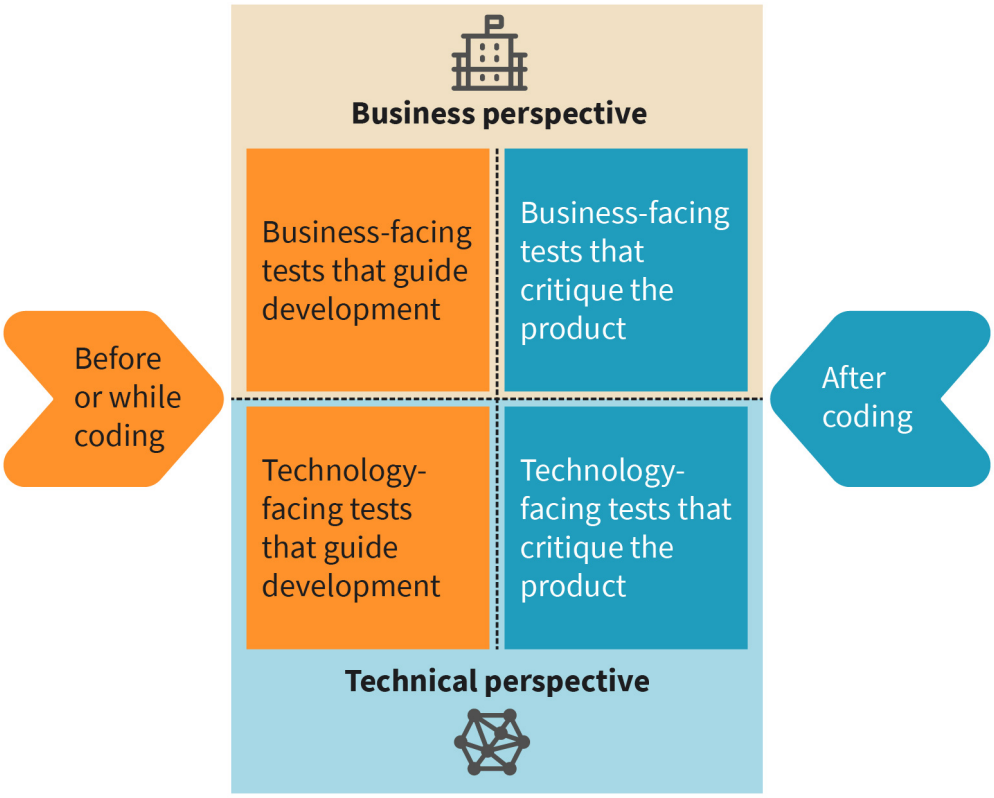
Agile testing also requires the user to rethink their contribution to the development process. In companies that mostly carry out traditional projects, it is common for users' involvement in the requirements elicitation process or in the evaluation of new software versions to be rare (but comprehensive). Because of the shorter release cycles, agile development approaches provide for almost continuous communication between the user and the development team. In the team, the users are represented by the product owner, who needs to be up to date with the latest developments in the business domain. Users should be prepared to provide feedback on early prototypes or new functions that is more frequent, yet narrower in scope. The feedback can be given in regular meetings (for example, the sprint review meeting) based on previously conducted tests.

Management should be aware of the fact that agile testing, particularly in its initial phases, is an investment. The introduction of employees to new ways of working, the effort needed to create tests; the procurement, installation, and maintenance of the necessary tools; as well as the provision of test data, initially only generate cost that is not fully covered by automation. As is the case with many investments, the investment only pays off later. In the case of agile testing, the main effect is minimization of risk when delivering the software, because it has been developed in close cooperation with end users and has been tested much more intensively.

5.2 Testing Levels and Agility

Agile testing breaks up the sequential processing of the individual testing levels. The individual testing levels still exist and still are necessary because they fulfill functions that are not replaced in agile development. However, they can be carried out in a flexible and incremental manner. This means that the team decides when and to what extent the testing levels are executed for certain parts of the software. The agile testing quadrants proposed by Gregory and Crispin (2019) can support the team in making these decisions. The model does not categorize test types by level; instead, it groups them according to two dimensions. As shown in the figure below, tests are differentiated according to whether they are business- or technology- facing (first dimension) and whether they guide development or test the product as a whole (second dimension).

Figure 42: Agile Testing Quadrants



Source: Sandra Rebholz (2023), based on Gregory & Crispin (2019, p. 59).

The first dimension (the vertical axis) distinguishes between business facing tests and technology-facing tests. Business-facing tests check the software according to business-related requirements, such as the correct calculation of insurance premiums. These tests must be conducted by team members with adequate domain knowledge (for example, end users). Technology facing tests verify the basic functionality of small software compo-

nents and the correct interaction between these components. They also check if user input is being processed at an adequate speed. These tests are carried out by team members with technical expertise (for example, developers or staff from IT operations).

The second dimension (the horizontal axis) distinguishes between tests that are mainly targeted at only testing the basic functionality (**happy path** testing) and tests that include non-functional requirements. Based on the number of functional tests passed, the team can measure their current progress with respect to implementing the requirements. This is why these tests are also said to guide development. If the tests are mainly targeted at testing non-functional requirements, as for example the usability or reliability of the software, or if the tests check unusual usage paths, the tests are said to critique the product.

Happy path
This is a test case that checks whether the system is working correctly when used as intended. Exceptional or error conditions are not considered.

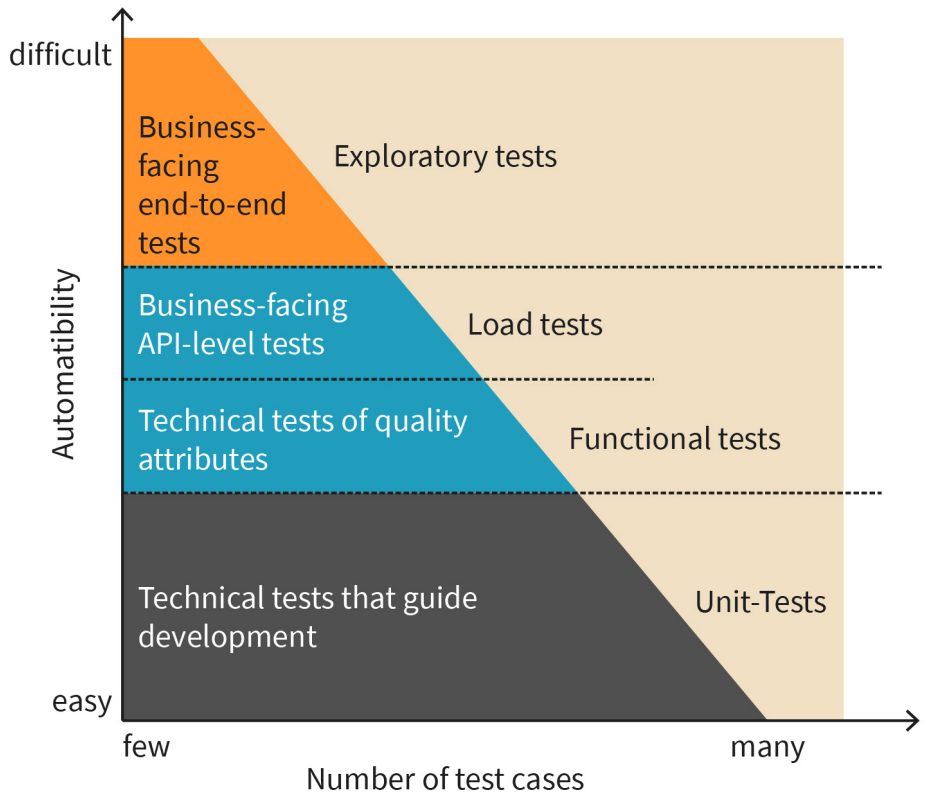
First Quadrant: Technology Facing Tests That Guide Development

Unit tests (or component or module tests) are technology-facing tests that check the basic functionality of small software components (units). Accordingly, they belong to the first quadrant of the agile testing quadrant model shown above. In object-oriented programming, the units are typically chosen on the level of individual classes.

Unit tests
Tests that check whether individual units of the source code work as expected.

Unit tests form the foundation of agile testing, especially when considering the number of unit tests as compared to the other test types. The test pyramid in the figure below shows the quantities of the different test types. The harder it is to automate a certain test type, the less test cases should be created. The reason for this rule is simple: Such tests have to be carried out manually and hinder agility due to the effort involved. However, this does not mean that these tests are less important. Experts are needed to choose the best test cases and optimize the efficiency of manual testing. Unit tests are typically executed and evaluated by using automation tools such as the unit testing framework JUnit. The developers write the necessary test cases themselves in their local development environment.

Figure 43: Targeted Number of Tests Based on Test Types

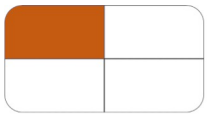
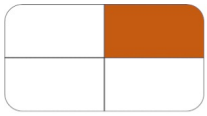


Source: Created for the IU (2023) based on Gregory & Crispin (2019, p. 66).

Unit tests are carried out early in the software development process and, in agile projects, these tests are executed after each change to the code base. In this way, the functionality of the application can be assessed early on. Each newly developed component that is to be integrated into a software system must pass a set of suitable unit tests before the integration. At the beginning of a project, this can be more complicated than expected. Each component has at least one interface with another component (for example, to receive required input parameters or to pass on results). However, many of these components do not yet exist in the beginning, and thus are not ready for use in the test. This is the reason unit tests require mock objects that simulate the required interfaces. If necessary, these mock objects must be created before the unit tests are actually executed, for example by using the mocking framework Mockito (Mockito Project, 2023).

Table 9: Characteristics of the Four Testing Quadrants

Quadrant 1	
Business or technology facing?	Technology facing
Guide development or critique the product?	Guide development

Type of tests	Unit tests
Tester's profile	Developer
Percentage of all test cases	80–90%
Degree of automation	Very high
Quadrant 2	
Business or technology facing?	Business facing
Guide development or critique the Product?	Guide development
Type of tests	Functional tests, examples, story tests, prototypes, simulations
Tester's profile	Domain expert (user or analyst)
Percentage of all test cases	5–15%
Degree of automation	Partly automated, partly manual
Quadrant 3	
Business or technology facing?	Business facing
Guide development or critique the Product?	Critique the product
Type of tests	Explorative tests, scenarios, usability tests, user acceptance tests, alpha/beta tests
Tester's profile	Experienced user ("power user")
Percentage of all test cases	1–5%
Degree of automation	Purely manual
Quadrant 4	
Business or technology facing?	Technology facing
Guide development or critique the product?	Critique the product

Type of tests	Performance and load tests, security tests
Tester's profile	Technologically knowledgeable testing expert
Percentage of all test cases	1–5%
Degree of automation	Supported by specific tools

Source: Created on behalf of the IU (2023).

In the approach of test-driven development, unit tests are an essential element of the development cycles that are executed several times a day. When the cycle is stopped and the software is passed on to integration testing depends on the degree of automation, the individual project plan, and the software process model framework applied.

Second Quadrant: Business Facing Tests That Guide Development

Testing can only reveal the existence of errors: it cannot ensure the absence of errors. It is impossible to guarantee one hundred percent error-free software. Instead, the team should at least make sure that they consider as many perspectives in the tests as possible. One of these perspectives is covered by business facing tests. They check the basic functionality of software based on business scenarios or stories, without deviating too far from the intended main flow through the application. Because these functions are mostly realized by several components in a cross-sectional manner, they can only be executed after partial integration of the software. Therefore, in classic testing, such tests are often executed in late integration stages, for example as part of system testing. Agile teams are faster and more flexible in this regard because they can coordinate their work on interdependent components. This makes it easier to agree on an early integration of components that are required for a specific functional test.

Functional tests can be initiated via an application programming interface (API), a graphical user interface, or batch processing. Tools for automating such tests vary depending on the mechanism applied, which can make these tasks challenging.

Third Quadrant: Business Facing Tests That Critique the Product

Business-facing tests assigned to the second quadrant aim at clarifying whether or not the current state of the software basically fulfills the required functionality from a business perspective. In contrast, business facing tests assigned to the third quadrant aim at leaving the happy path and have professional users execute unusual test cases instead. They put the software through its paces and draw upon well-known problem patterns that they are familiar with due to their expertise. In addition to known special cases, they also test non-functional requirements related to the end user's perspective, such as usability. Because such exploratory tests obviously cannot be automated, there should only be a manageable number of these tests.

In classic processes, these tests are carried out in the form of acceptance tests and executed only late in the process. In agile teams, these tests can be executed very early on based on a preliminary version of the software. Such early feedback sometimes reveals new requirements for future versions that would have otherwise been identified much too late and would have caused high costs.

Fourth Quadrant: Technology Facing Tests That Critique the Product

Software can also be subjected to this kind of high-level testing from a technical perspective. This involves testing non-functional requirements mainly arising from IT operations. These tests typically include load and performance tests, security tests, and reliability tests. These technology facing tests usually require complex tools that monitor the system under high load conditions and that record all observed errors. They must be carried out by experienced technicians (for example, by security specialists) in sophisticated test environments. Consequently, the number of test cases should be reduced to what is necessary.

In traditional development, IT operations are usually only involved when deploying the software to the production environment. However, if the team only realizes at this point that the software cannot be operated with the available resources, expensive changes to the architecture are required (such as replacing a slow software library). Developers in agile teams already take technical requirements into account when designing and executing unit tests. However, it is difficult for them to anticipate certain interdependencies. These interdependencies may only become obvious in certain contexts, for example problems due to poorly configured server settings may only be apparent in the context of the integrated system.

5.3 Test Automation

In order to ensure that frequent changes to the software do not lead to decreased confidence in the software's functionality, regression tests must be carried out after every change. This effort is only possible with the help of automating test execution and test documentation activities. In this section, we will look at methods and tools that enable the automation of the test types introduced above.

Automating Unit Tests with JUnit

Fortunately, automated unit tests are in frequent use today. The goal of unit test automation is the highest possible test coverage of the instructions and control structures in the program code. Unit tests are open-box tests that aim at testing all statements, conditions, and paths of a system component, depending on the concrete test case requirements. Typical requirements to automated unit tests are reproducibility, independence, and the capability to detect regression.

Reproducibility

Each unit test should be fully reproducible. This is achieved through a fixed set of test data in the test case that does not depend on the successful execution of other unit tests.

Independence

Furthermore, there must not be any functional dependency between unit tests: The sequence in which unit tests are executed should not influence the test result. Each test must individually ensure that the necessary preconditions are established. This also includes, for example, setting up the database with all the necessary data before executing the test on this data. If another test is executed afterwards, all changes to the database and the internal state of the system must be rolled back so that the next test can be executed in the original system state.

Detecting regressions

Evolutionary software development processes are characterized by adding functions to the system incrementally. Such processes must ensure the required degree of quality of the released software version at the end of an iteration. With respect to unit tests, this implies that all unit tests that already have been passed successfully, must also be passed after the software has been changed. Tests that make sure that previous functionality is retained after changing the program code are called regression tests. Thus, when finishing an iteration, all existing and new unit tests must be passed successfully. For this reason, test automation is an important prerequisite for evolutionary software development.

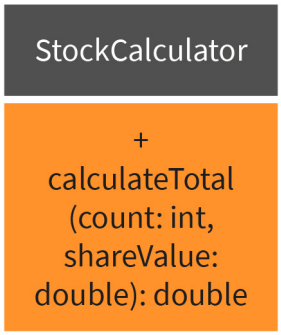
The testing framework JUnit

There are a number of tools for about 80 programming languages that support the creation, execution, and documentation of unit tests. One of the most widely used tools for Java is JUnit. It is part of the **xUnit** family of unit testing frameworks and can easily be integrated into the development environment via plugin (for example, in Eclipse).

xUnit testing frameworks
This is a set of unit testing frameworks that have a common structure and have become standard in the software engineering community. JUnit is a popular example of this type of testing framework.

In the following section, the basics of unit testing with JUnit are explained based on a simple example. The example (based on the example described in Beck (2003, p. 3–11)) makes use of a stock calculator, which calculates the total stock value for a given stock price and a given number of shares.

Figure 44: UML Class Diagram for the Stock Calculator



Total Value = Count * Share Value

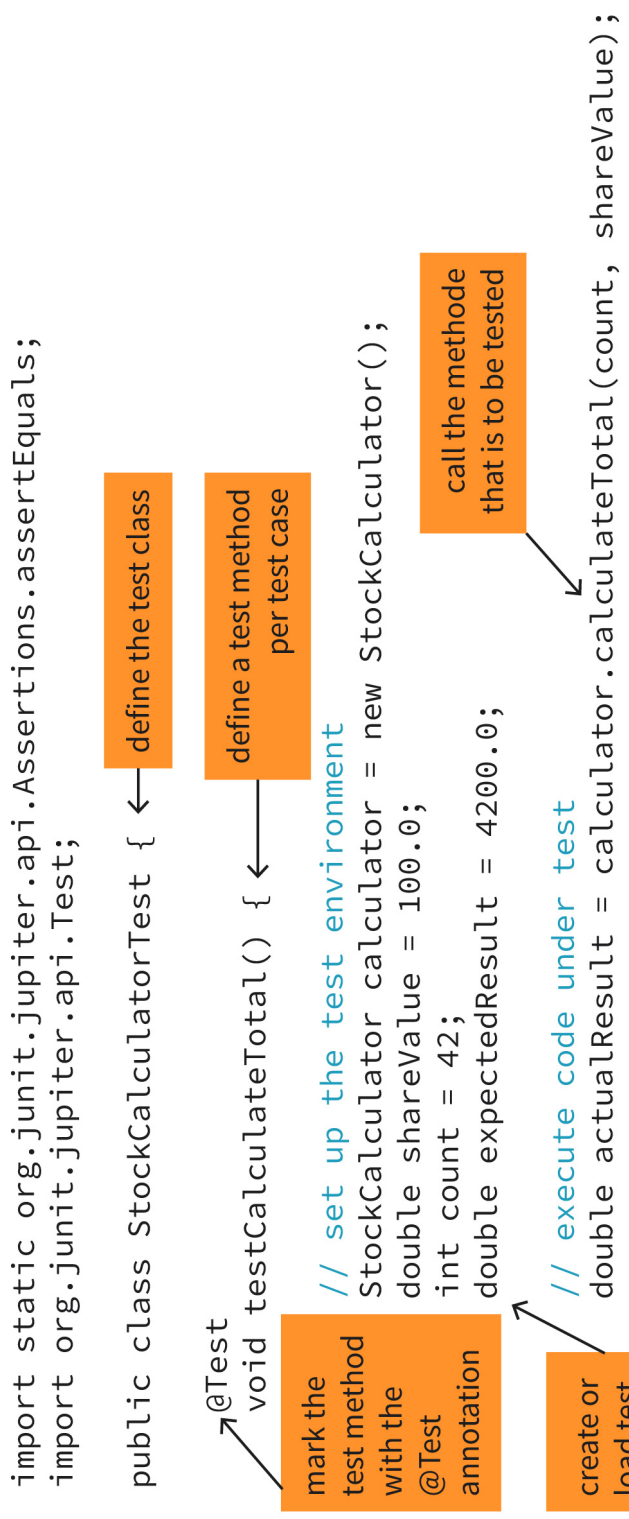
Source: Sandra Rebholz (2023).

JUnit tests are implemented in the Java programming language, typically in the same development environment or in the same development project as the production code that is to be tested. In order to separate production code and test code, the unit tests are put in their own package. For every class being tested, there is exactly one JUnit test class. The name of the test class typically corresponds to the name of the class to be tested appended by "Test." In the example of the class `StockCalculator`, the test class is then named `StockCalculatorTest`. The methods of the test class implement the individual test cases. They start with the annotation `@Test`as shown in the figure below.

JUnit Assertions
An assertion method is a utility method to check whether a certain condition holds true in a test.

Each test method implements exactly one test case. First, the test method implementation loads or defines the test data (here: `shareValue` and `shareNumber`). Then, the test is executed by calling the method under test (here: `calculateTotal`). Finally, the returned result is compared with the expected result using **JUnit Assertions**. Assertions are predefined methods such as `assertTrue`, `assertFalse`, or `assertEquals`, that allow for easy comparison of the actual result with the expected result. In the example, two floating point values are compared to each other specifying a delta of 0.0001 for which both numbers are still considered equal.

Figure 45: JUnit Test Class for the StockCalculator class



Source: Sandra Rebholz (2023).

Assertions are an important element of test automation in Java. They indicate the information that is used to automatically evaluate whether a test case is successful or not. When executing all the test cases of a test class, the test driver (in this case JUnit) uses this information to provide the most precise feedback possible to the developer on whether the software works as expected.

As soon as a test class contains more than one test case, the developer should consider using specific lifecycle methods for preparing before and cleaning up after the test cases have been executed. In JUnit 5, there are four types of life cycle methods that are marked by the annotations `@BeforeAll`, `@BeforeEach`, `@AfterEach`, and`@AfterAll` (Bechtold et al., 2022). The annotation`@BeforeEach`marks the method that implements all setup activities that are to be performed before a test case is run. JUnit automatically calls this method before each test case. In the same way, JUnit automatically calls the method marked with the annotation `@AfterEach` after a test case has been run. This method can be used to release system resources or to perform other types of cleanup operations. The methods marked with the `@BeforeAll` or the `@AfterAll` annotation are executed before or after all test cases are run in the current class. Implementing and using the JUnit 5 life cycle methods is optional and can be omitted if they do not provide any added value (as in the example code shown above).

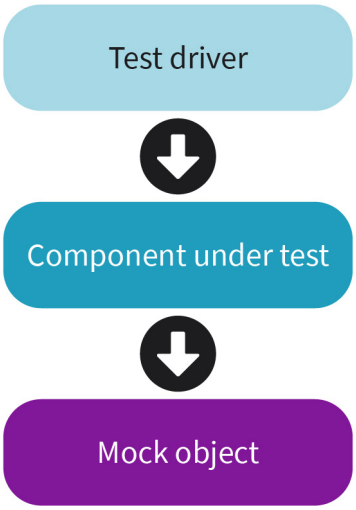
Automating Integration Tests with Mock Objects

In contrast with traditional approaches, the integration of software takes place early in agile approaches. As soon as at least two software components have been completed, they can be integrated into one system. In integration tests, the interaction of these components is tested (during or after integration) in order to check whether or not the components work together as specified.

Integration tests differ from unit tests in that they do not focus on the internal behavior of individual components, but rather, on the communication between components using their technical interfaces. There are two aspects that must be addressed by these tests: the functions provided by the interface, and the data that is exchanged via the interface. In integration testing, test cases that explicitly focus on testing the functions provided by the components' interfaces as comprehensively as possible need to be created and executed. In particular, this involves testing how the interfaces behave if they are not invoked correctly or receive erroneous data. Apart from designing test cases with a focus on interfaces, integration tests do not fundamentally differ from unit tests in the way they are created or executed. Therefore, integration tests can also be automated using JUnit. The only difference here is that components are not tested as strictly isolated units, instead, the test uses several components in combination.

Because agile integration tests are carried out early in the development process, there are many components that are not yet available – especially in the beginning. In these cases, missing components must be simulated by placeholder objects, drivers and mock objects, in order to enable early testing. The figure below illustrates the interaction between drivers and mock objects.

Figure 46: Test Drivers and Mock Objects



Source: Created on behalf of the IU (2023).

Drivers are software fragments that simulate the invocation of other components. Often, JUnit test cases are drivers themselves. In contrast, mock objects simulate components that are called by other components. In integration testing, mock objects and drivers are used to simulate technical interfaces with external systems. This is because only the finished software system can be connected to external systems, but the interaction of the interfaces must be tested beforehand.

For the automation of integration tests, it is advisable to use specific tools to create or code these mock objects. Mockito is a widely used mocking framework(Mockito Project, 2023). Provided that the interface description already exists, only one line of code is necessary to describe the behavior to be simulated by the mock object (see figure below). Thus, mock objects pretend to implement the given interface. The configuration of the mock object is performed in the test case implementation, but not in the production code. Accordingly, there is no need to reset any changes in the source code after the testing has been finished, which would require some effort and would be prone to errors.

The code shown below demonstrates how to use Mockito mock objects to implement an automated integration test for the StockCalculator example. The example code is extended to include a data access layer. A data access object will be created for the StockCalculator class according to the data access object (DAO) pattern (see Sun Microsystems, n.d.). Imagine a scenario in which the developer in charge of the implementation of the DAO class has not yet finished the work but has already published the interface description (*IShareDAO*). Based on this interface, a mock object can be created that simulates the behavior of the data access layer (in this case, retrieving stock data from the data base; see figure below).

Figure 47: Using Mockito to Create Placeholder Objects (Mock Objects)

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.mockito.Mockito.*;

public class StockCalculatorTest {
     StockCalculator calculator;
     IShareDAO shareDAO;

     @BeforeEach
     public void setUp() {
          calculator = new StockCalculator();
          shareDAO = mock(IShareDAO.class);

          when(shareDAO.getPrice("Alphabet")).
          thenReturn(100.0);
          when(shareDAO.getCount("Alphabet")).thenReturn(42);
     }

     @Test
     void testCalculateTotal() {

          // specify expected result
          double expectedResult = 4200.0;

          // execute code under test
          double actualResult = calculator.calculateTotal(
               shareDAO.getCount("Alphabet"),
               shareDAO.getPrice("Alphabet")
          );

          // verify the result
          assertEquals(expectedResult, actualResult, 0.0001);
     }
}
```


Source: Sandra Rebholz (2023).

Using Mockito like this means that the test methods do not have to be adapted once the interface is completed. Only the three lines of code for creating and specifying the intended behavior of the mock objects in the setup method have to be replaced by creating an object of the actual implementation class of the interface.

Automating System and System Integration Tests

As the agile team progresses with the integration of software components, system or system integration testing becomes necessary. These tests no longer check individual components but focus on integrated systems and applications in their context. Thus, placeholders no longer replace interfaces between components; instead they stand in for interfaces between systems. The principle is similar, and they can be automated with the technologies described above, however, they are carried out in different production-like environments.

Automating business facing tests

When performing system and system integration tests, the testing is often carried out via the GUI. In such cases, automation tools like Selenium or SikuliX are used instead of xUnit frameworks, such as JUnit. These tools generate scripts that reproduce user interactions with the software in a repeatable way. In this way, some manual GUI tests can be automated.

The scripts for automating GUI tests can be created in two different ways: the test scripts can either be coded by the developers directly, or they can be recorded by the tester via the graphical user interface. In the latter case, **"Capture & Replay"** tools are used for recording the user interface actions.

Capture & replay
This type of tool records manual user interactions via the graphical user interface and uses the recording for future testing of the application.

Although this sounds promising and simple, GUI tests are rather complicated in practice because the scripts cannot really cope with changes in the GUI. They only work correctly if the referenced user interface elements are actually found when "re-playing" the tests. Each change to the GUI can potentially cause the complete script (and thus, the complete test case) to fail.

At this high level of integration, tests are strongly based on domain-related test cases. Consequently, test automation requires too much effort compared to its added value and should be replaced by manual exploratory tests carried out by experts (see testing quadrants two and three).

Test Data Management

In introductory literature, the underlying concepts of test automation are often described using fairly simple examples. However, if looked at in detail, test automation can represent a big design problem: The test cases also contain the test data and thus, test cases and test data are hard-wired (as in the stock service calculator example shown above). Mixing test cases and test data brings the risk of high maintenance when it comes to the test cases. Professional testing in larger projects involves the ongoing evolution of the software and the maintenance of several hundreds of test cases, potentially causing high costs. Imagine, for example, that the customer ID of the test customer that is used in many tests has changed. If the test data was defined at the beginning of each test case in a transparent way, this change could probably be addressed automatically by a small program

(for example, a shell script). If the customer ID is different in different parts of the testing code, or if it is otherwise hidden, several hours of test data maintenance must be invested on a regular basis.

This problem can be solved by establishing test data management that separates the test data from the test cases by storing the data in specific tables and linking it to the corresponding test cases. Because nowadays, each programming language offers a suitable library for reading files in, for example, CSV format, accessing the test data from the testing framework is straightforward and easy.

SUMMARY

In this unit, the main characteristics and concepts of traditional testing were outlined. Agile testing differs from traditional testing in terms of its main principles. This also has consequences in terms of the organizational requirements of agile testing.

Agile testers do not rely on pre-set schedules in order to decide which type of test is adequate in the current situation. They aim to gain knowledge about the uncertainties that have to be resolved in the project. Depending on whether these uncertainties are more technology- or business-related, and on if they are grounded in functional or non-functional requirements, the correct type of test can be chosen based on the agile testing quadrant model by Gregory and Crispin (2019).

In order to manage the many tests despite short release cycles, agile teams rely on tools for test automation and for test data management. Depending on the progress of component integration, the degree of automation decreases or technical tests are replaced by manual tests that assess business functions. If these functions are initiated via the GUI, specific tools can be used that allow the scripting and automatic replay of UI interactions.

UNIT 6

AGILE DELIVERY AND DEPLOYMENT

STUDY GOALS

On completion of this unit, you will be able to ...

- understand the concept of continuous delivery and how it differs from continuous build, continuous integration, and continuous deployment.
- identify the individual phases of the continuous delivery pipeline.
- describe the methods and tools that are applied in each phase of continuous delivery.

6. AGILE DELIVERY AND DEPLOYMENT

Introduction

Agile software development processes (particularly those that involve short release cycles) require a high degree of test automation. Besides the various test types and the testing levels, the need for automation also affects many other steps in the software development process. Accordingly, this unit will focus on ways to automate or at least optimize the entire development process.

The continuous delivery approach involves defining a pipeline consisting of various methods and tools for software process automation and optimization. This approach encompasses a number of related concepts, such as continuous build, continuous integration, and continuous deployment.

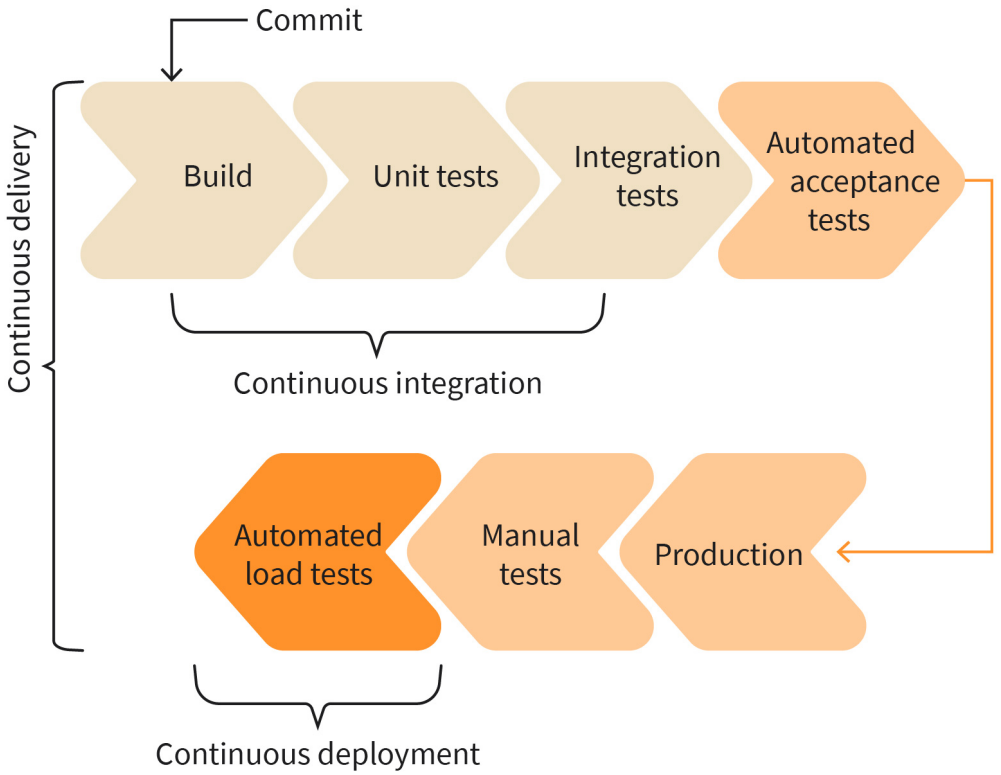
6.1 Continuous Delivery Pipeline

Continuous delivery is a key aim outlined in the agile manifesto: “Our highest priority is to satisfy the customer through early and continuous delivery of valuable software” (Beck et al., 2001).

As the name suggests, the term “continuous delivery” encompasses concepts, methods, and tools that enable the ongoing conveyance of software to the customer.

Although there is no consensus on a precise, detailed definition of continuous delivery in the literature on the subject, most authors refer to one essential element: the continuous delivery pipeline. According to the literature, the pipeline encompasses a largely standardized sequence of steps that start with changing the code base (commit), followed by mostly automated integration and test phases, and ending with the installation and configuration in the production environment (deployment), as depicted in the figure below.

Figure 48: The Continuous Delivery Pipeline



Source: Created on behalf of the IU (2023).

At first, this pipeline may seem very similar to the traditional software process. However, there are some important differences. Firstly, the batch size in agile projects is much smaller. This means that the developer introduces, for example, a new function that they have developed within a sprint, but not a whole component or a whole application. Secondly, developers who lack experience with agile projects might be surprised by the order of the tests. After all, the automated acceptance tests are not carried out as the final step before the software is taken into operation as is common practice in standard testing approaches (Hambling, et. al., 2019), but directly after integration.

Advantages of the Continuous Delivery Pipeline

The effort required to establish continuous delivery in an enterprise is significant. After all, the entire software production chain must be supported by tools, testing environments, and other infrastructure so that even the smallest code change can be compiled, integrated, tested, and deployed automatically. This approach brings a number of advantages.

Time-to-market

Any delay in the production process could mean a competitive disadvantage. If a competitor is faster to come out with innovative features, there is one less reason to buy the software. Of course, this does not only apply to pure software companies, but also to IT departments in digital companies where innovative business ideas can only be brought to market with the aid of suitable software.

Risk minimization

Sometimes, it is not the speed of product delivery that is most important. In such cases, continuous delivery offers the following advantage: error-prone, manual processes for software rollout are replaced by automated or optimized approaches that are more reliable. When continuous delivery is applied, each release is significantly smaller and thus, more manageable. The process is carried out more frequently, is more automated and more thoroughly tested, making it more reliable, reproducible, and less prone to software regressions. In sum, all these aspects significantly lower the risk of a faulty deployment.

Fast feedback

The more a software process relies on planning in advance, the more the software is created based on assumptions. As a consequence, the team only gets feedback on whether the software is of sufficient quality and meets the actual customer requirements late in the development process. However, if a company invests in a continuous delivery pipeline, the developers obtain feedback after each commit. This enhances the team's confidence that the software meets the functional and non-functional requirements at any point in time, and the team gets the confirmation that they have done the right thing. This allows the developers to focus on implementing new features instead of worrying about the next software delivery that might be scheduled only in a few months.

Continuous Delivery Principles

The individual steps and measures that form part of the continuous delivery approach are based on a well-known saying in the agile developer community: "If it hurts, do it more often!" At first, this does not seem to make much sense. After all, pain is generally seen as something to be avoided.

Let us take a look at how this principle might be applied to the software development process. In a traditional process, a developer might work according to a long-term plan for as long as possible. They might want to delay the insights that come with starting the integration, testing, or the deployment to the production environment until later in the process,delaying the results might be waiting for them. By doing so, deployment becomes a process that often brings unpleasant surprises and becomes a risky operation for both developers and IT operations staff. The continuous delivery pipeline is designed in direct contrast to this approach and is based on three principles: regularity, traceability, and regression detection.

Regularity

A central principal of continuous delivery is the regular execution of software delivery processes, particularly the activities that involve risks. Regularity enables the team to work professionally. Obviously, this principle also requires a high degree of automation in order to reduce the effort for recurring tasks. This applies specifically to the automated provision of different environments for testing, integration, and deployment.

Traceability

If new software versions are created with high frequency, the people involved in the development process quickly lose track of the changes that have been made to the software or its configuration in a certain version of the system. If you want to reproduce a certain error condition, it is not only necessary to use the exact same software but also to reproduce the execution environment as accurately as possible. This can only be achieved if the software version and the corresponding version of the test environment and its configuration are documented. Because continuous delivery automates the test environment setup based on installation scripts, the effort for versioning these environments is reduced to the versioning of the installation scripts.

Regression detection

Any change to a software system can introduce errors to parts of the code that are not directly related to the change. Sometimes, the changes even affect completely different systems that communicate with the changed software via interfaces. Such unpleasant errors (which, paradoxically are only introduced by improving the software) are called regressions.

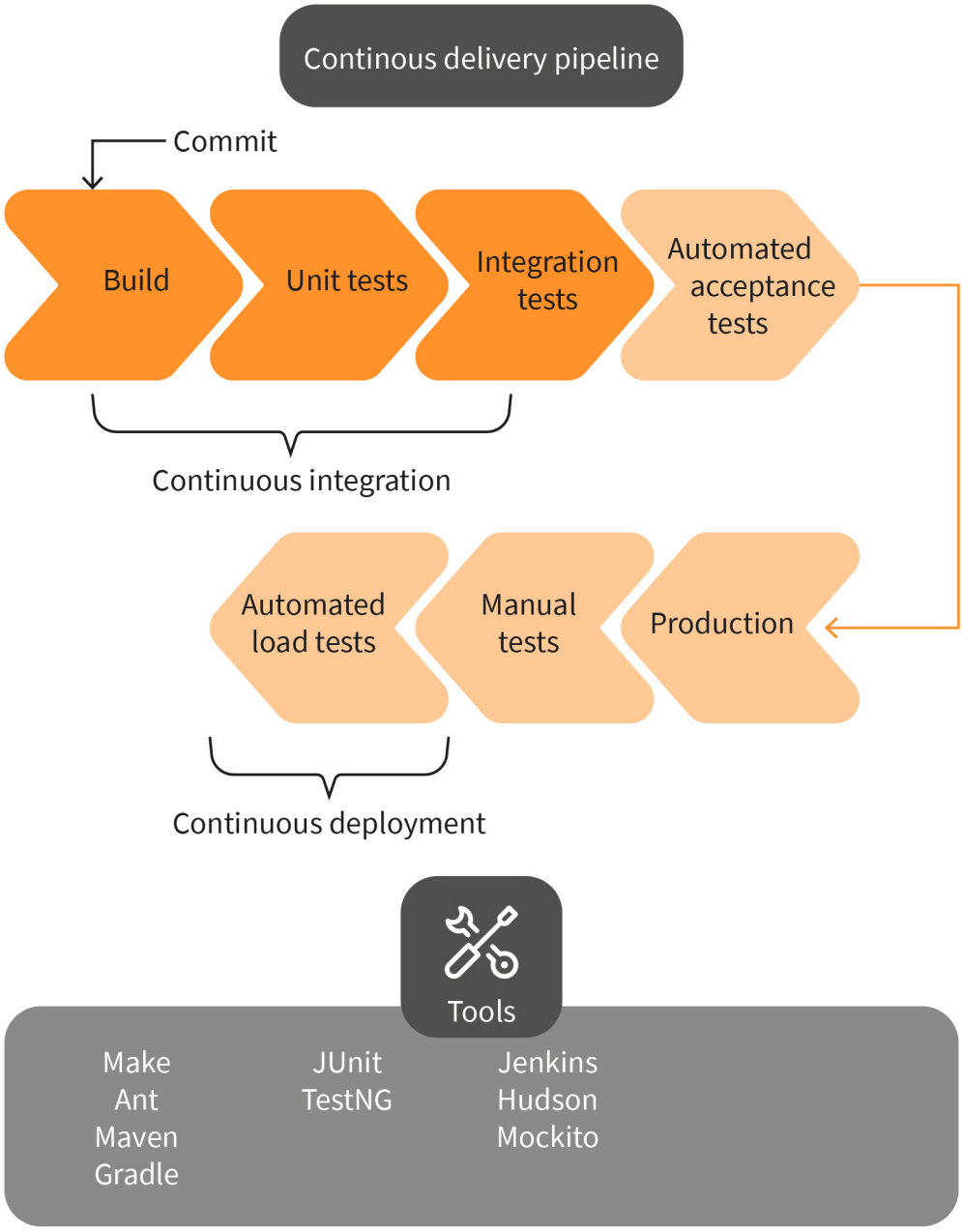
Regressions occur in all projects – regardless of whether they are carried out in the traditional or in the agile way – and they can only be detected through regression tests. Such tests repeat all available test cases for the software system, including those that do not seem to be related to the change at first glance.

In traditional testing, regression tests simply represent an additional type of test that has to be carried out once for each release. However, when applying the approach of continuous delivery, regression testing is a regular occurrence. Therefore, supporting the development team with regression testing is an essential principle of continuous delivery.

6.2 Continuous Build and Continuous Integration

The first steps of the continuous delivery pipeline consist of the build, unit testing, and integration testing. This section introduces the methods and tools that support the automation or optimization of these steps. The figure below illustrates how this range of tools maps to the individual steps in the pipeline.

Figure 49: Initial Steps of the Continuous Delivery (CD) Pipeline (Including Tools)



Source: Created on behalf of the IU (2023).

This section will introduce the continuous build concept and the tools needed for its application. Following this, the approach of continuous integration will be explained in more detail.

Continuous Build

The method

Continuous delivery aims at accelerating the entire software delivery process. In order to achieve this goal, as many steps as possible are automated, or at least optimized. In fact, there are many companies where the automation of these steps is already a reality. For example, automating the build process is a standard procedure in software development and there are plenty of sophisticated tools that support this process. Although the concrete steps can vary depending on the programming language, the following refinement of the continuous delivery pipeline is typically applied in a continuous build process:

1. Compile the source code (if necessary load any dependencies, such as program libraries)
2. Load required resources
3. Execute unit tests
4. Create a basically deployable software package and publish the package in a package repository (note that the package will usually need further testing)

The configuration of the continuous build tool determines whether these steps are triggered with each commit made by the developer. For example, it is common practice for only changes to the main branch to be considered relevant for the continuous build process. Depending on the version control system that is used, these branches are called "trunk" or "main." Only changes to those branches initiate the automated build process. The continuous build process can also be configured in a way that leaves it up to the developer to decide up to which point the build process should execute after a commit. However, all variants of the continuous build process described above offer the possibility to automatically compile, test, and generate a deployable package after each code change.

Many developers will already know some of these automation steps without having consciously applied the continuous build method. For example, in the well-known integrated development environment, Eclipse, every save operation of the source code initiates a compile operation.

The tool set

Applying the approach of continuous build in a project requires a specific set of tools. However, the need for such tools is not new and almost as old as the idea of decomposing software into components in order to reduce complexity. For example, the make tool was developed in 1977 and represents an early version of today's build automation tools. It is a tool written in the C programming language and is primarily used for compiling program libraries on the Unix platform (Insitute of Electrical and Electronics Engineers & The Open Group, 2018).

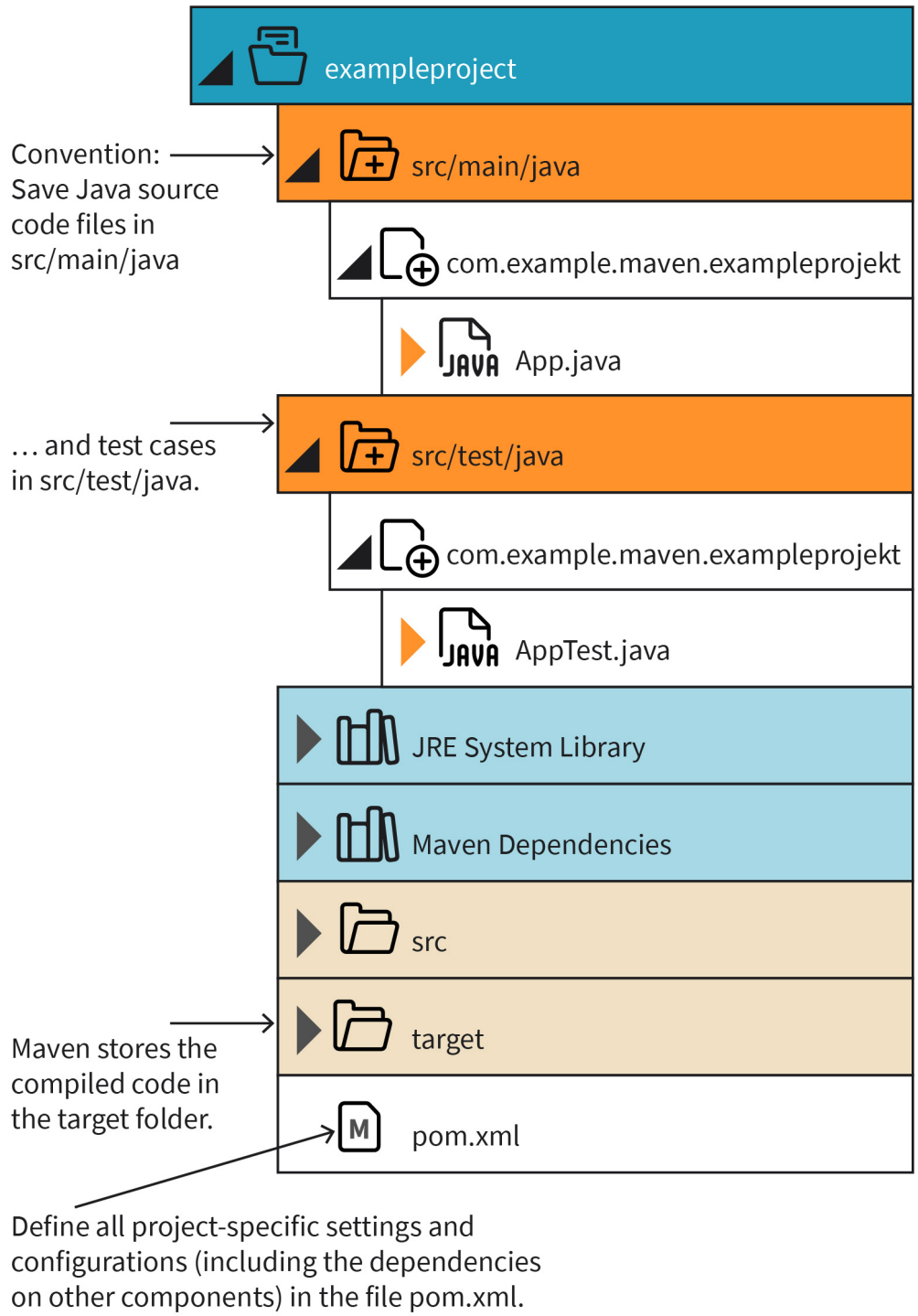
However, with increasing project size, the Makefiles that are used for configuring the compile process become very complex. The build tool Ant (which is based on Java and uses XML files for configuration purposes) was developed at the end of the 1990s to address

this problem. In comparison to the structure of Makefiles, the XML-based structure of Ant configuration files offers significant advantages with respect to the platform-independent storage of build configurations (Apache Ant Project, 2022).

Nevertheless, Ant also has a significant disadvantage: Although the XML files are quite similar in different projects, the developers must specify all configurations. This contradicts the important software engineering principle of avoiding redundancies (commonly known as the DRY principle: don't repeat yourself). That is why, a couple of years after the release of Ant, Maven was developed in order to get rid of unnecessarily overloaded configurations.

At that time, many programming languages, programming libraries and development tools were greatly simplified through a shift in the paradigm that aimed at configuring or programming only such aspects that do not conform to a pre-defined convention (“convention over configuration”; Varanasi, 2019, p. 4). Integrating these conventions as part of the development environment relieves the developer of learning the conventions by heart. If, for example, a developer wants to generate a new project in Eclipse that uses build automation based on Maven, they simply select the corresponding project type. Then, the development environment takes care of setting up the correct directory structure based on the corresponding convention (as shown in the figure below).

Figure 50: Directory Structure of a Maven Project



Source: Created on behalf of the IU (2023).

Maven refines the continuous build process described above by defining the Maven default life cycle (Apache Maven Project, 2023). This life cycle is a Maven-specific process that is made up of no less than 23 individual steps, starting with various validation and initialization steps. For example, the life cycle may begin with checking if all source files are structured according to the convention and if all specified resources are available. This is followed by the actual compilation process and the tests and continues up to the deployment on the chosen application server (as depicted in the figure below). By specifying one of the individual steps, the developer can control up to which point this life cycle is to be executed.

Table 10: The Core Steps of the Maven Default Life Cycle

validate	Checks the project structure and validates that all necessary information is available
compile	Compiles the sources (for example, using the Java compiler)
test	Runs the unit tests
package	Creates packages from the compiled code (for example, as JAR or WAR)
integration-test	Deploys the created packages into the integration environment
verify	Checks whether all necessary output files (and their contents) were created as expected
install	Adds the created package to the local Maven repository
deploy	Publishes the final package to a remote repository

The life cycle steps are realized by plugins that are integrated in Maven. On the console, status information showing the functionality of the plugins is printed out after the process is started. For example, if the compile step is invoked for the first time, the corresponding plugin prints out a message that changes have been detected and the compilation process has been started. If this step is repeated without changing the program, the plugin omits the unnecessary re-compile (as shown in the figure below).

Figure 51: Example Output on the Console From Maven

```
[INFO]
[INFO] -----------------------< com.example:exampleproject >------------------
[INFO] Building exampleproject 0.0.1-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
...
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ exampleproject
---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory [...]/exampleproject/src/main/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ exampleproject ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 3 source files to [...]/exampleproject /target/classes
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  1.399 s
[INFO] Finished at: 2023-03-11T12:57:38+01:00
[INFO] ------------------------------------------------------------------------
```

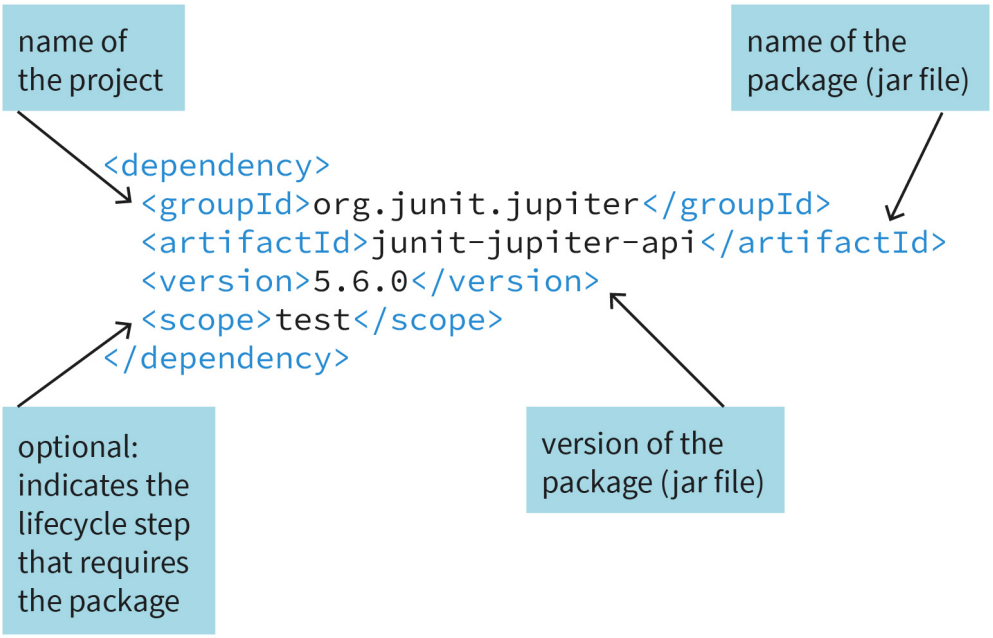
```
[INFO]
[INFO] -----------------------< com.example:exampleproject >------------------
[INFO] Building exampleproject 0.0.1-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
...
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ exampleproject ---
[INFO] Nothing to compile - all classes are up to date[INFO] -----------------
...
```

Source: Sandra Rebholz (2022).

All project-specific information that cannot be determined and automatically processed using the Maven convention must be defined in an XML file called pom.xml. In a very simple project, this file is almost empty. But if the project requires external libraries, the pom.xml needs to be extended. As shown in the figure below, integrating JUnit requires six lines of configuration code.

The configuration code only contains information that is necessary for integrating the required library into the project. Obligatory information includes the name of the project responsible for publishing the package (*groupId*), the package name (name of the jar file), and the required version (*version*). If the library is required only for a certain step of the lifecycle, this step can be specified in the scope element of the XML file. In the example of JUnit, the corresponding life cycle step is “test.” Based on this information, Maven can initiate a fully automated build process including the download and installation of all required files in the appropriate versions from an online repository containing readily compiled libraries ("binary repositories"). For developers, this facility alone makes the development process significantly easier.

Figure 52: Annotated Dependency Declaration in the pom.xml



Source: Sandra Rebholz (2023).

Another widely used build automation tool is Gradle (Gradle Inc., 2023). Gradle offers the possibility to extend the pre-defined steps of the system by custom steps, making it more flexible than Maven. Fortunately, Maven and Gradle are compatible: They both use the same directory structure and support the popular binary repositories such as the Maven Central Repository. Using these repositories allows one to easily find, download, and integrate external packages. However, Gradle is based on the programming language Groovy. Consequently, using Gradle in one's project requires a basic knowledge of this programming language.

Continuous Integration

The continuous build process essentially refers to all tasks that can be automated within the integrated development environment (IDE). Many of the steps a developer has to pay attention to are supported by the tools presented above. But how can the process of integrating the work of the whole team or even of several teams be automated or optimized?

The method

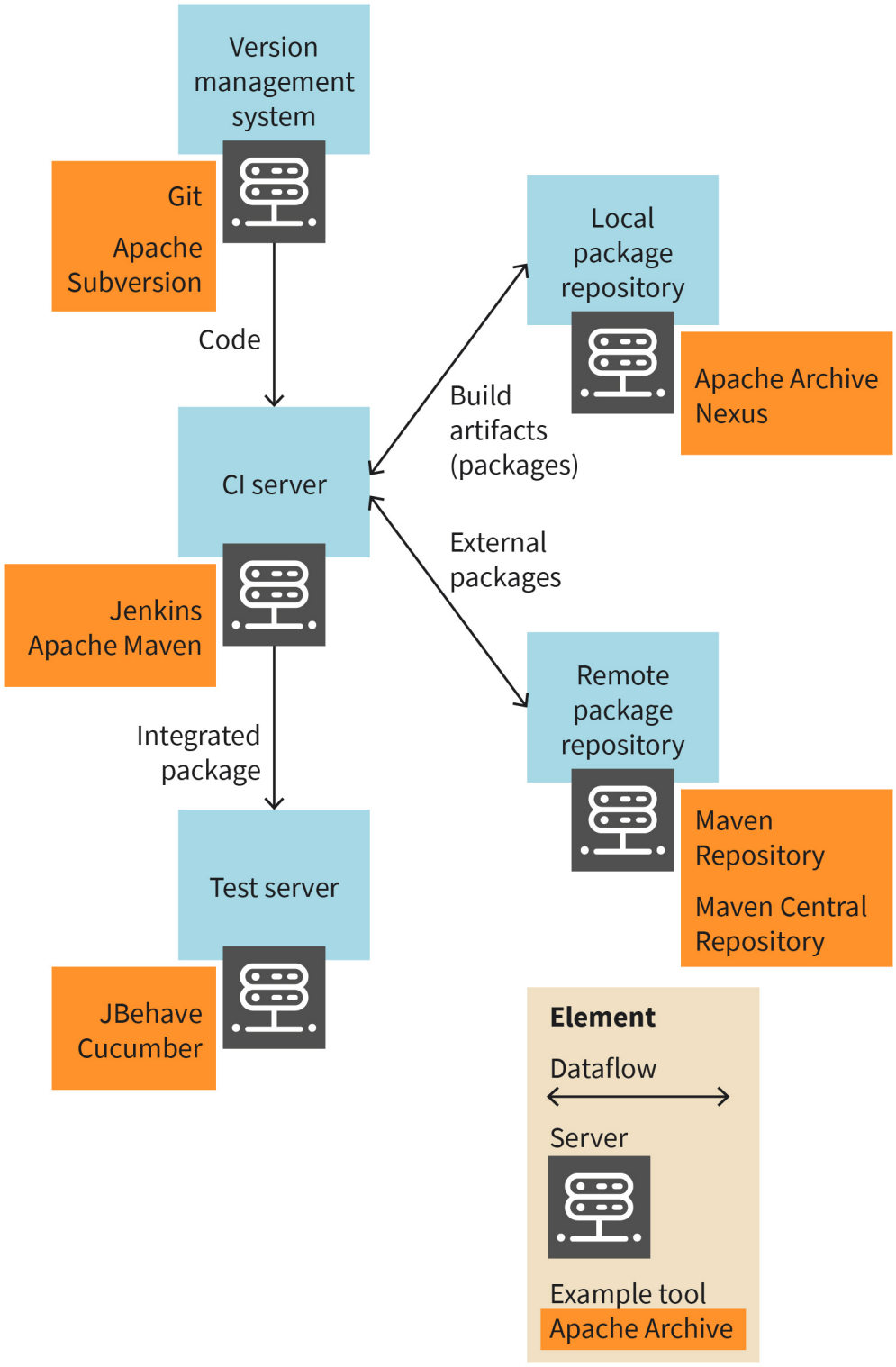
A closer look at the Maven life cycle reveals a promising step called "integration test." This step does not refer to the actual integration test, but to the deployment of the software on the integration server. And it is precisely at this point that the transition from continuous build to continuous integration takes place. For the latter, a shared build server is used to integrate components that have been implemented by different developers. This shared build server is also called a continuous integration (CI) server. The server initiates the inte-

gration process as soon as the specified trigger event fires (for example, after a certain period of time, or after a commit to the global repository of the version management system). The following four steps are executed as part of the integration process:

1. Download the latest code base from the version management system
2. Download all required external libraries from the remote package repository
3. Compile and integrate the whole system into one deployable unit
4. Deploy the created binary files on a server for further testing (e.g. load and acceptance tests)

A continuous integration environment contains a number of additional servers that are connected to the CI server in order to be able to automate all the steps described above. With the exception of the remote package repository, all servers are usually operated locally within the company.

Figure 53: Systems and Dependencies in a Continuous Integration Setup



Source: Created on behalf of the IU (2023).

A CI server offers a lot of flexibility with respect to the configuration of the integration process (also known as the continuous integration pipeline) described above. For example, the pipeline can be configured to include steps for an automatic database setup. In addition, manual release processes can be configured for specific steps, so that they are only executed after manually releasing them. Among other things, this allows for explicitly controlling which version of the integrated software is passed on to the subsequent steps of the continuous delivery pipeline.

The tool set

The main tool for automating and optimizing the integration process is the CI server. There are many CI servers on the market. One popular CI server is Jenkins (Jenkins Project, 2023), which defines all steps of the build and integration process as "jobs." Each job has its own workspace that stores all used and generated files. Jenkins offers the central advantage that is compatible with both the established version management systems such as Git and Apache Subversion (SVN), and with Maven. More specifically, a Jenkins job can be configured to use Maven to automatically compile and test the assembled components. All that needs to be done is to specify the Maven command along with the corresponding pom.xml. In principle, the build process on a CI server is identical to the automated build process in the local development environment, the difference being that the CI server uses the integrated components of different developers to perform a common compile.

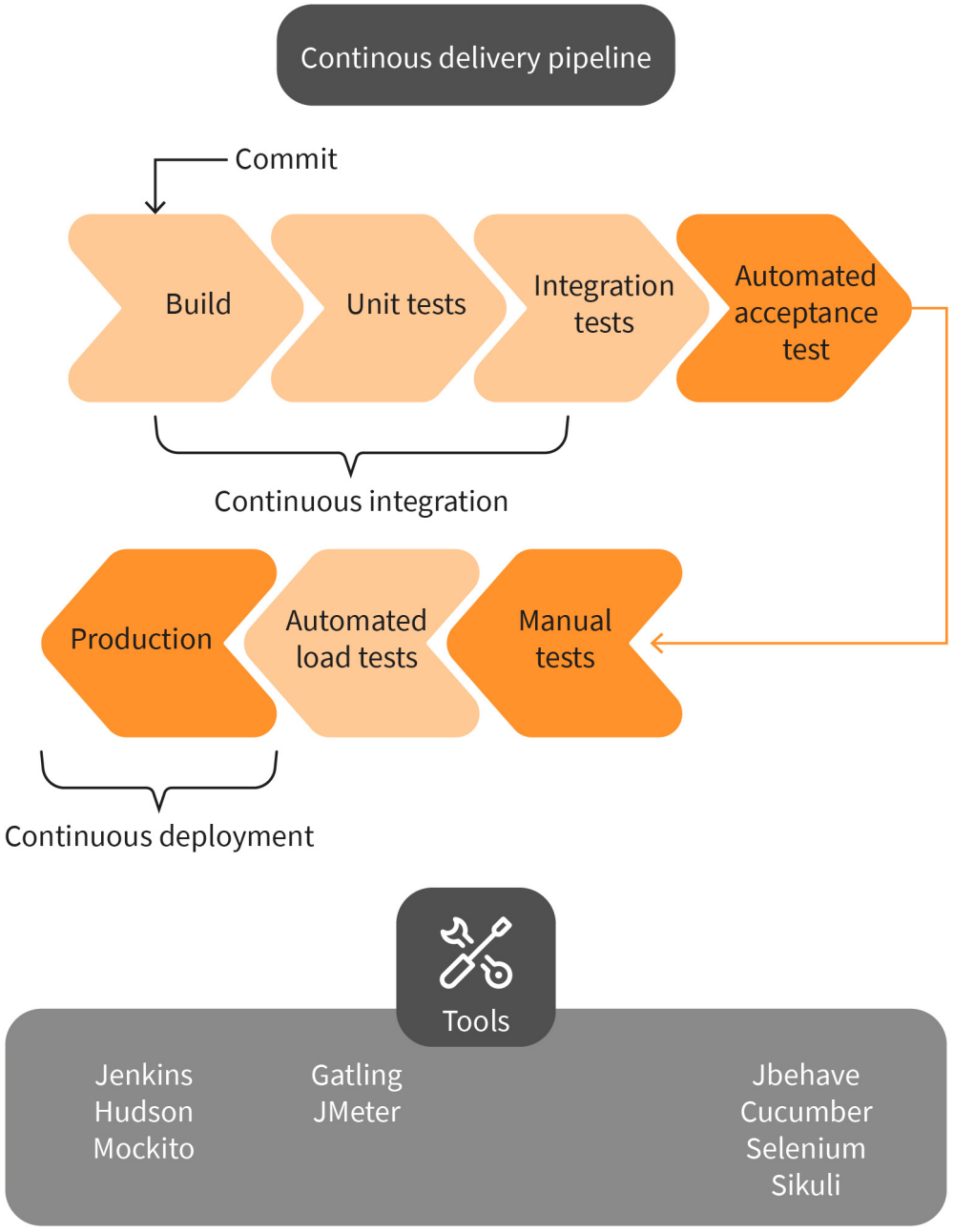
The functionality of Jenkins can also be extended using plug-ins, which are provided and maintained by an active developer community. The variety of plug-ins provide a rich set of additional functions such as the automatic adjustment of environment variables depending on the current step in the build process, the versioning of complex job configurations, or easy-to-use GUIs for managing individual build pipelines. If the existing plug-ins do not fulfill the project-specific requirements of the continuous integration process, Jenkins can also be extended by custom plug-ins.

In addition to Maven, Jenkins also supports other build tools such as Ant and Gradle. Due to its wide distribution, Jenkins offers interfaces to many relevant tools that are widely used in the software engineering community. Such tools include integrated development environments such as eclipse, IntelliJ, and Visual Studio Code, and bug tracking systems such as Redmine, Bugzilla, Trac, and Jira.

6.3 Acceptance Tests, Load Tests, and Continuous Deployment

As soon as the CI server (for example, Jenkins) has assembled and compiled a new release of the software, subsequent tests can be carried out. According to the continuous delivery pipeline, these tests involve automated acceptance and load tests that are carried out before exploratory testing. In terms of the agile testing quadrants according to Gregory and Crispin (2008), the test types in quadrants two and four are carried out here.

Figure 54: Final Steps and Tools in the CD Pipeline



Source: Created on behalf of the IU (2023).

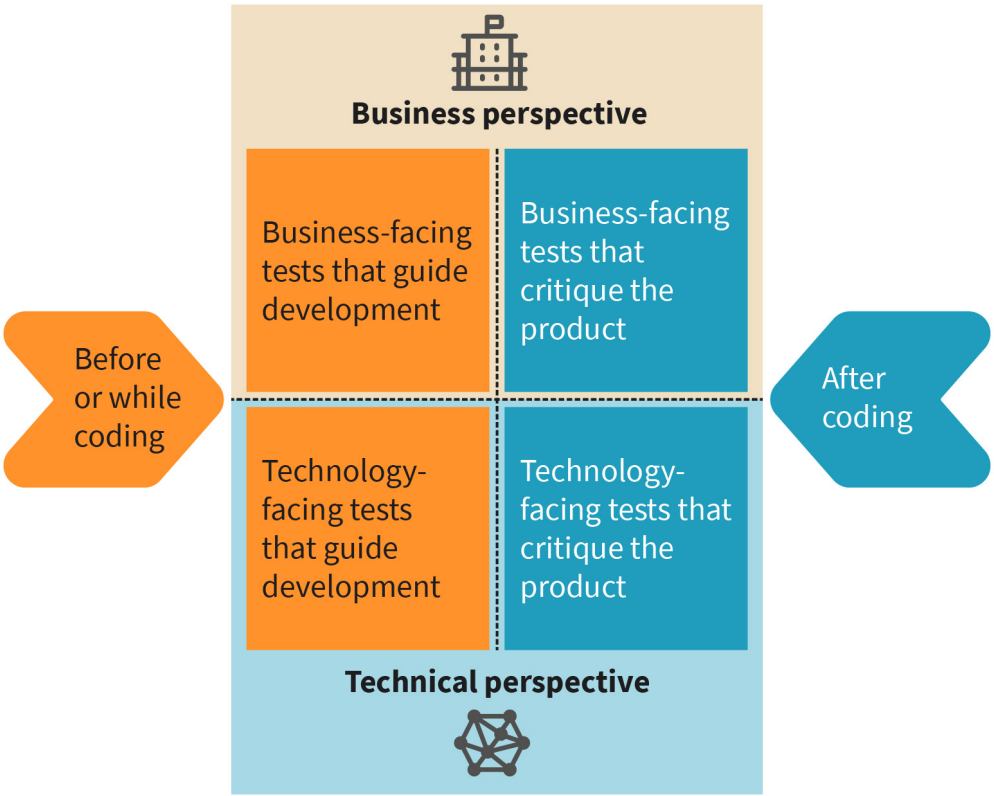
First Make it Run – Then Make it Fast!

At first it may seem odd that acceptance tests are not carried out at the end of the continuous delivery pipeline, just before putting the software into operation. After all, acceptance tests are the tests that prove that all the requirements were realized correctly. The key concept behind the placement of the acceptance tests before the automated load tests is

the agile principle of “first make it run, then make it fast.” According to this principle, an agile team should focus on implementing the basic functionality first before targeting non-functional requirements such as performance. This is one of the main reasons automated acceptance tests are carried out before load and performance tests in the continuous delivery pipeline.

It is also important to remember that the continuous delivery pipeline does not follow the typical ordering of test phases followed in traditional testing. Instead, the phases are ordered according to the agile testing quadrants. Based on this model, classic acceptance testing is divided into an automated part, which verifies basic domain-related functions (quadrant 2), and a manual, exploratory part that is carried out by experienced users or customers (quadrant 3) at the end of the continuous delivery pipeline.

Figure 55: Agile Testing Quadrants



Source: Sandra Rebholz (2023), based on Gregory & Crispin (2019, p. 59).

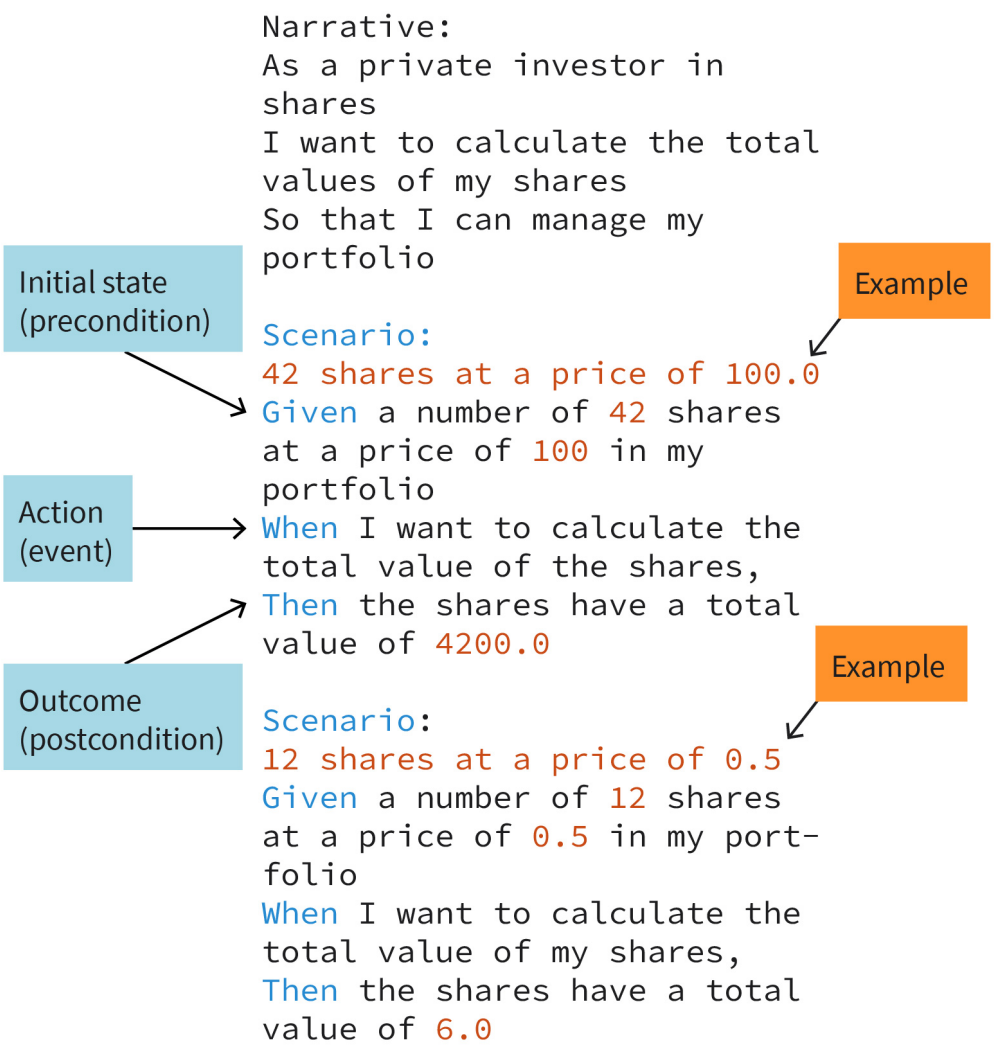
Of course, the automated part should cover as many acceptance criteria defined by the customer as possible. In order to achieve this goal, tools are used that allow customers to be involved in the specification of test cases. Nevertheless, these tests do not replace the formal acceptance procedure of the software by the customer as required by legal contracts.

Automating Acceptance Tests

By definition, automated acceptance tests are not intended to be carried out by the customers themselves. Nevertheless, the underlying test cases should be specified in collaboration with the customer. However, this brings about a dilemma: typically, the customer does not have any experience in the specification of test cases, not to mention programming languages for developing automated tests. For this reason, automated acceptance tests use tools that allow customers to specify test cases in very simple language. Some tools even use languages that are very similar to natural language. This approach is also referred to as **Behaviour-Driven Development** (BDD); (North, 2006). BDD extends the basic idea of test-driven development in that customers describe the expected behavior of the software in natural language, which serves directly as a test case specification. A widely used BDD tool for Java is JBehave. In JBehave, the customers define a user story by means of the language Gherkin, which has a very simple syntax targeted to support BDD. User stories are defined in the form of individual scenarios which again are described by some lines of text (as shown in the figure below).

Behaviour-driven development
This is a technique in agile software development that specifies test cases in a simple textual format with a focus on the intended behavior of the software.

Figure 56: Describing a User Story in JBehave



Source: Sandra Rebholz (2023).

A scenario describes an example of how the software should behave in a certain context. In order to describe this context, the initial state (or precondition) of the scenario is described by “givens”, a term that is also used as a key word (“Given”) in JBehave. A “given” is a textual description of a precondition indicating concrete values for the example at hand. Based on these lines, the developers of automated tests know precisely how to set up the code before running the test.

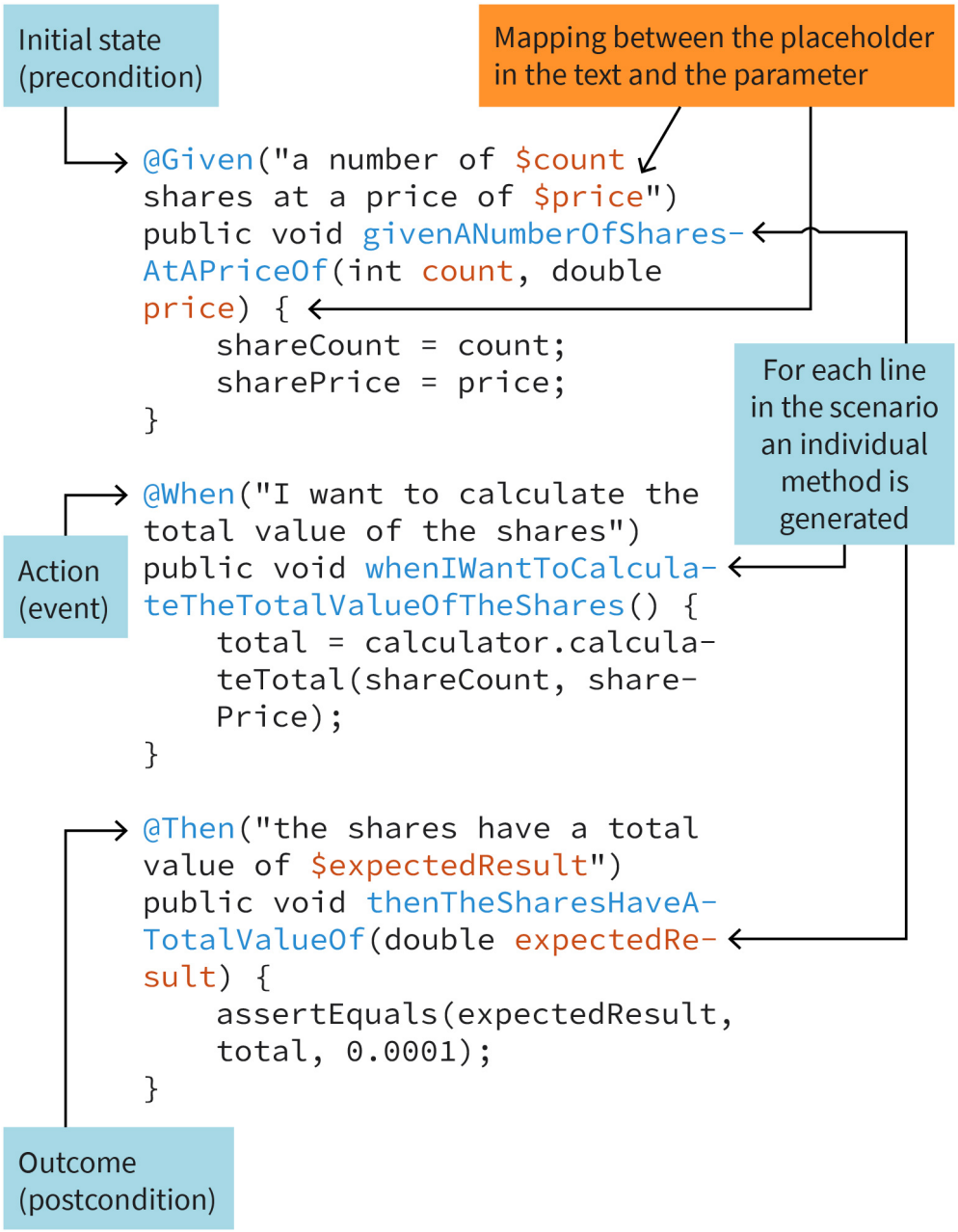
In addition to the preconditions, the action (or event) that is carried out in the scenario has to be described. The event is introduced by the key word “When” followed by a textual description of the user’s action. The event represents a domain-related action that is explicitly supported by the software. Based on the textual specification, the developer knows what function is required by the customer and how to invoke it when running the test.

Finally, the key word “Then” concludes a scenario. Here, the expected outcomes (or post-conditions) of the scenario are described in a textual format. Again, concrete values for the example should be indicated and should match the previously specified preconditions. However, at this point, the developers have to rely on the domain competence of the customers.

The specification of acceptance tests in BDD resembles the basic structure of unit tests. When specifying a unit test, you begin by setting up the test case. Next, you run the test, and then, you verify the actual results. This similarity in structure is not a coincidence, but a proven design principle used for specifying tests. Following this three-step process ensures that all relevant details are considered and that the test cases are more readable.

As soon as the customer has completed a few stories in cooperation with a developer or test expert, the developers can start implementing automated test cases in Java code. For each scenario, a test class containing suitable methods for each line in the test case specification must be generated. Similar to JUnit, JBehave uses annotations based on the Given-When-Then keywords as described above. This helps to trace the assignment between the individual steps in the scenarios and the test functions on and enables test automation. Because JBehave knows the semantics of these annotations, it can derive the correct invocation sequence without further involvement of the developer.

Figure 57: Mapping Steps From the JBehave Scenario to Java methods



Source: Sandra Rebholz (2023).

In the code snippets shown above, JBehave appears to recognize the numerical values in a story. In order for this to work, JBehave must know that certain words in the text refer to example data. For this purpose, the developer must add parameters to the generated methods and assign them to the text. JBehave offers the use of placeholders that are prefixed by the $ sign (as shown in the figure above). If they are named in the same way as the parameters, JBehave can automatically derive the correct assignment. The resulting

advantage is significant: By means of the stories, the customer can specify almost any amount of additional test data without having to adapt the program code or any configurations.

Note that the example shown above implements the same test as the JUnit test case introduced before. The only difference is that it was not conceived by the developer (aside from the refinement using JBehave) but was created based on the customer's story. Again, this reveals the advantage of automated acceptance tests. If the customers write their own tests, they can be sure that in the end the system will behave according to their own specifications. The risk of developers implementing and testing domain-related functions based only on assumptions (due to their lack of domain knowledge) is significantly reduced.

Load Testing

As soon as a code change has passed the automated acceptance tests, the next step along the continuous delivery pipeline is initiated. According to the principle "first make it run, then make it fast," the testing activities now focus on non-functional requirements such as the system's behavior under load. This ensures that the system still performs adequately when it is used by a large number of users. In classical software engineering, setting up an infrastructure for load testing is a complex and error-prone task which is why it is usually rarely carried out. In a continuous delivery pipeline, however, automatic build tools are used, meaning that such infrastructures can be prepared much more easily and reproducibly. Additionally, there are tools, like Gatling, that can be used as a driver for automated load tests, for example, as the final step in the Jenkins build pipeline.

Gatling simulates a given number of users by replaying a recorded sequence of interactions in a web application in parallel. The basic procedure is quite simple and consists of the following four steps:

1. Record user interactions with the recorder app.
2. Verify, and if necessary, adapt the simulation generated by the recorder.
3. Run the simulation.
4. Check the result.

Recording user interactions

Recording user interactions is very convenient because it only requires starting the recorder and then navigating through the application as a customer would do. As a prerequisite for recording, the browser settings must be configured to use a local proxy server that is executed in the background by the recorder. During the recording, predefined scenarios (ideally developed in collaboration with the customer) are executed. Upon completion of a scenario, the recording is stopped and Gatling generates an installation script containing individual instructions for each user interaction. The script is located in a specific Gatling folder sorted by simulation (for example, *user-files**simulations*) and can be opened using a standard text editor.

Adapting the generated script

The developer can then check the generated instructions and if necessary, adapt or supplement them. This adaptation process requires a basic understanding of the object-oriented and functional programming language Scala, which is why we will not go into greater detail here. However, it is important to know that recurring patterns in the scenarios can be defined as individual components "objects," that can be reused in different scripts. This approach is recommended for complex projects.

Running simulations

In Gatling, recorded load test simulations are run via the command line. This makes it easy to integrate and automatically start a load test as part of the Jenkins build pipeline. Gatling provides a web-based report for the testers which contains comprehensive statistics based on the test results.

Analyzing the test results

The web-based report can be viewed easily in the web browser and contains information on response time, throughput and other details. Analyzing the test results can provide hints as to which concrete components of the application do not perform as expected and slow down the whole process.

A tightly integrated tool chain is very important for the successful implementation of a continuous delivery pipeline. The need for manual operations between the individual steps can thereby be reduced to a minimum. Jenkins, Gatling, and Maven are examples that allow for this kind of seamless integration by the means of suitable plugins. Using the Gatling plugin for Maven (as shown in the figure below) provides an easy mechanism to integrate Gatling tests in the Maven life cycle by a specific step for Gatling tests (Gatling Corporation, n.d.-b).

Figure 58: Adding a Gatling Test to the Maven Life Cycle in the pom.xml

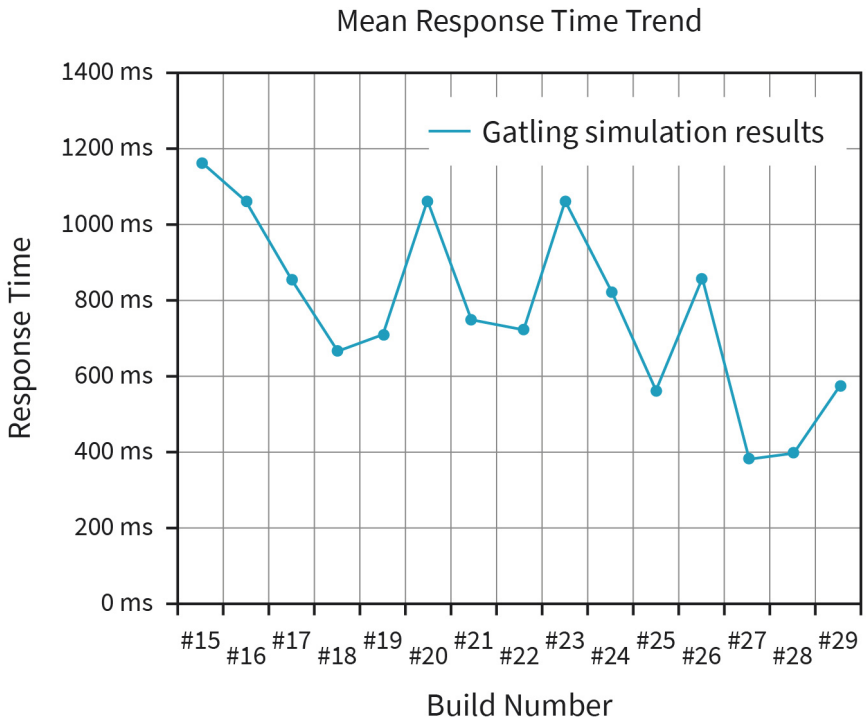
```
<dependencies>
  <dependency>
    <groupId>io.gatling.highcharts</groupId>
    <artifactId>gatling-charts-highcharts</artifactId>
    <version>x.y.z</version>
    <scope>test</scope>
  </dependency>
</dependencies>

<plugin>
  <groupId>io.gatling</groupId>
  <artifactId>gatling-maven-plugin</artifactId>
  <version>x.y.z</version>
</plugin>
```

Source: Created on behalf of the IU (2023).

This step (like any other Maven build step) can then be used in Jenkins to control the build process. What is more, there is a Gatling plug-in for Jenkins that provides useful overviews visualizing the results of the simulations run at the end of the build pipeline (this is shown in the figure below).

Figure 59: Visualizing the Mean Response Time Trend Across Builds



Source: Sandra Rebholz (2023) based on Jenkins Project (n.d.).

Tools for automated load tests can usually only be applied to web applications. Even though the number of web applications in companies is increasing, it is not possible to use these tools for testing the entire application landscape. One of the few exceptions is Apache JMeter which can also be used to test databases. Because automation is not optional, but absolutely necessary in the continuous delivery pipeline, in the worst case, special programs have to be written in order to perform load tests. They do not necessarily have to consider the user interface, but they can directly call functions of the business layer in order to test the system under load.

After load testing, manual tests (as stated in testing quadrant 3) are carried out. They are done by experts and aim at testing the system in holistic scenarios that focus primarily on non-functional requirements from the business perspective. This involves, for example, testing the usability from the end user's point of view. Because such tests are highly dependent on expert knowledge, they cannot be automated, and therefore, do not fit into the scheme of agile testing. For this reason, we will not cover manual tests in detail. However, if you are interested in learning more about this topic you can refer to Pfahl,et. Al., 2014).

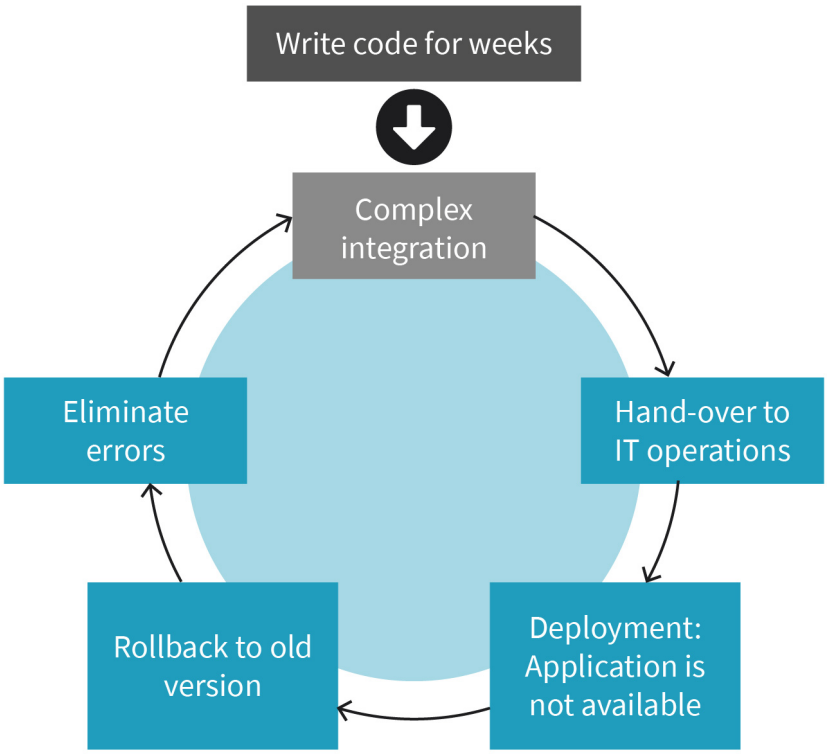
Continuous Deployment

After covering test types from all four testing quadrants, the last step in the continuous delivery pipeline is the deployment to production, which is the provision of the updated software in the productive environment. At first glance, this step is nothing more than

another deployment step in addition to the previous deployments to the different test and integration environments. However, in practice, there are important differences that mean that this step is rarely automated as part of a continuous deployment pipeline.

Automation during continuous deployment carries a business risk that does not exist during the other deployments. A failed deployment in the integration server is less harmful than in the production system where “real” customer data is processed and a loss of data or limitations in the availability of the system can have a direct negative impact on the business. Continuous deployment defines concepts and methods that mitigate these risks with the aim of implementing a delivery pipeline that is fully automated (apart from manual testing). One approach from classic software development is the establishment of a rollback process that always keeps a functioning version of the software as a backup and reinstalls this version if the automatic deployment fails (as shown in the figure below).

Figure 60: Vicious Circle of Classic Deployment



Source: Created on behalf of the IU (2023) based on Snyder (2013).

A disadvantage of this method is that important information about the errors, like log files, is often lost with the rollback. The aim of continuous deployment is to avoid this problem by greatly decreasing the scope of a change (when compared with classic deployment) while at the same time greatly increasing the deployment frequency. This reduces not only the complexity but also the risk of a single release. It might even be more advantageous to simply overwrite an erroneous release with the following release instead of carrying out a costly rollback. This approach is known as roll-forward.

Another challenge comes from the fact that in the productive environment there are significantly larger data sets than the test data sets in the test environments. Adding additional fields to a database, for example, leads to extensive update procedures in the database management system. This can involve all kinds of problems that also occur in data migration projects. For example, in the event of an error, the previous state of the database has to be restored as quickly and reliably as possible. In addition, during migration, any database access must be prevented or handled intelligently.

The concept of continuous delivery also includes methods to address these problems and to act in an agile manner with respect to the deployment into the production environment. For example, a database can be treated like an individual component. Changing the database can then be tested and rolled out separately before deploying the actual software that requires the change in the database. This leads to a significant decrease in complexity, which also reduces risk. Additionally, each execution of the continuous delivery pipeline is clearly identified with a version number. The current state of the code base as well as all configurations, including the database schemes, are then summarized under this version number. This makes it easier to trace erroneous changes to the database scheme.

SUMMARY

Approaches such as continuous build and continuous integration, automated acceptance and load tests, and continuous deployment enable the entire development process to be automated and optimized. Together, these steps form the continuous delivery pipeline and represent a necessary extension of the previously presented measures of test automation.

The build automation tool Maven and the Jenkins build pipeline are essential tools to support these steps and enable an automated integration of components verified by unit tests. Concepts such as behavior-driven development (BDD) support the automation of business facing tests. JBehave is an example of a BDD tool that can be applied in practice.

This unit demonstrated how to test software under high loads using the simulation and load testing tool Gatling. The challenges of automating the deployment of software into production were also discussed, and methods were presented for addressing these issues in the continuous delivery approach.

BACKMATTER

LIST OF REFERENCES

Agile Alliance. (2022). *Agile glossary*. https://www.agilealliance.org/agile101/agile-glossary/

Anderson, D. J. (2011). *Kanban: Evolutionäres change management für IT-Organisationen* [Kanban: Evolutionary change management for IT organizations]. dpunkt.verlag.

Apache Ant Project. (2022). *Apache Ant*. https://ant.apache.org/

Apache Maven Project. (2023). *Introduction to the build lifecycle*. https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

Armour, P. G. (2000). The five orders of ignorance. *Communications of the ACM*, *43*(10), 17 – 20.

Bauman, C. (1958). Accuracy considerations for capital cost estimation. *Industrial & Engineering Chemistry*, *50*(4), 55A–58A. https://doi.org/10.1021/i650580a748

Bechtold, S., Brannen, S., Link, J., Merdes, M., Philipp, M., de Rancourt, J., & Stein, C. (2022). *JUnit 5 user guide* [Version 5.9.2.]. https://junit.org/junit5/docs/current/user-guide/

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, A., Kern, J., Marick, B., Martin, R. C., Mellor, S., Scwaber, K., Sutherland, J., & Thomas, D. (2001). *Manifesto for agile software development*. https://agilemanifesto.org/

Beck, K. (2003). *Test-Driven development by example*. Pearson Education.

Cohn, M. (2007, May 6). Advice on conducting the Scrum of Scrums meeting. *Mountain Goat Software*. https://www.mountaingoatsoftware.com/articles/advice-on-conducting-the-scrum-of-scrums-meeting

Cohn, M. (2022). User story. *Mountain Goat Software.* https://www.mountaingoatsoftware.com/agile/user-stories

Cunningham, W. (2009, February 15). *Debt metaphor* [Video]. YouTube. https://www.youtube.com/watch?v=pqeJFYwnkjE

Duvall, P. M., Matyas, S., & Glover, A. (2007). *Continuous integration. Improving software quality and reducing risk*. Addison Wesley Signature Series.

Fowler, M. (2019, May 21). *Technical debt*. https://martinfowler.com/bliki/TechnicalDebt.html

Gatling Corporation. (n.d.-b). *Maven Plugin*. https://gatling.io/docs/gatling/reference/current/extensions/maven_plugin/

Gatling Corporation. (n.d.-a). *Reports*. https://gatling.io/docs/gatling/reference/current/stats/reports/

Gradle Inc. (2023). *Gradle Build Tool*. https://gradle.org/

Gregory, J., & Crispin, L. (2019). *Agile testing condensed: A brief introduction*. Leanpub.

Hambling, B., Morgan, P., Smaroo, A., Thompson, G., & Williams, P. (2019). *Software testing: An ISTQB-BCS certified tester foundation guide* (4th ed.). BCS, The Chartered Institute for IT.

Hruschka, P., Lauenroth, K., Meuten, M., Rogers, G., Gärtner, S., & Steffe, H.-J. (2022). *Handbook RE@Agile. Version 2.0.0.* IREB International Requirements Engineering Board e.V.

Institute of Electrical and Electronics Engineers & The Open Group. (2018). *The open group base specifications*. IEEE & The Open Group. https://pubs.opengroup.org/onlinepubs/9699919799/

International Organization for Standardization. (2011). *Systems and software engineering – Systems and software quality requirements and evaluation (SQuaRE)–System and software quality models* (ISO Standard No. 25010:2011). https://www.iso.org/standard/35733.html

Jaguar MENA. (2014). Der Ingenium-Motor von JLR [The Ingenium engine from JLR] [Image]. *Wikimedia Commons*. CC BY 2.0. https://de.wikipedia.org/wiki/Ingenium-Motor

Jenkins Project. (n.d.). *Gatling plugin usage*. https://plugins.jenkins.io/gatling/#plugin-content-gatling-plugin-usage

Jenkins Project. (2023). *Jenkins*. https://www.jenkins.io/

King, J. (2010). *Estimation toolkit: Some useful techniques*. Kings Insight. https://kingsinsight.files.wordpress.com/2010/09/estimation-toolkit-june-2010-v1-0.pdf

Kneuper, R. (2018). *Software processes and life cycle models*. Springer Nature.

Larman, C., & Vodde, B. (2009). *Scaling lean & agile development: Thinking and organizational tools for large-scale Scrum*. Addison-Wesley.

Lawrence, R., & Green, P. (n.d.). *The humanizing work guide to splitting user stories*. Humanizing Work. https://www.humanizingwork.com/the-humanizing-work-guide-to-splitting-user-stories/

Lawrence, R. (2020). *How to split a user story*. Agile For All. https://agileforall.com/wp-content/uploads/2020/12/Story-Splitting-Flowchart.pdf

Metoews, I. (2016). A hex dump of the 318 byte Wikipedia favicon, or W. The first column numerates the line's starting address, while the * indicates repetition [Image]. *Wikimedia Commons*. CC BY-SA 3.0. https://en.wikipedia.org/wiki/Binary_file

Mockito Project. (2023). *Mockito*. https://site.mockito.org/

Monden, Y. (1994). *Toyota production system. An integrated approach to just-in-time*. Springer. https://doi-org.pxz.iubh.de:8443/10.1007/978-1-4615-9714-8_2

North, D. (2006). *Introducing BDD*. Dan North and Associates. https://dannorth.net/introducing-bdd/

Oxford University Press. (2022). *Emergence.* Oxford Learner's Dictionary of Academic English. https://www.oxfordlearnersdictionaries.com/definition/academic/

Patton, J. (2014). *User story mapping. Discover the whole story, build the right product*. O'Reilly.

Pfahl, D., Yin, H., Mäntylä, M. V., & Münch, J. (2014). How is exploratory testing used? A state-of-the-practice survey. *ESEM '14: Proceedings of the 8th ACM/IEEE international symposium on empirical software engineering and measurement* (pp. 1–10). Association for Computing Machinery.

Pohl, K., & Rupp, C. (2015). *Requirements engineering fundamentals: A study guide for the certified professional for requirements engineering exam foundation level / IREB compliant*. Rocky Nook.

Project Management Institute, Inc. (2022). *Introduction to disciplined agile*. https://www.pmi.org/disciplined-agile/introduction-to-disciplined-agile

Röpstorff, S./Wiechmann, R. (2012). *Scrum in der Praxis. Erfahrungen, Problemfelder und Erfolgsfaktoren* [Scrum in practice: Examples, problem areas and success factors]. dpunkt.verlag.

Rubin, K. S. (2013). *Essential scrum: A practical guide to the most popular agile processes*. Pearson Education, Inc..

Scaled Agile, Inc. (2021a, March 11). *Agile release train*. https://www.scaledagileframework.com/agile-release-train/

Scaled Agile, Inc. (2021b, February 10). *Innovation and planning iteration*. https://www.scaledagileframework.com/innovation-and-planning-iteration/

Scaled Agile, Inc. (2021c, February 10). *PI planning*. https://www.scaledagileframework.com/pi-planning/

Scaled Agile, Inc. (2021d, February 10). *Welcome to Scaled Agile Framework® 5!* https://www.scaledagileframework.com/about/

Scaled Agile, Inc. (2021a, March 11). *Agile release train*. https://www.scaledagileframework.com/agile-release-train/

Schwaber, K., & Sutherland, J. (2020, November). *The definitive guide to Scrum: The rules of the game*. The Scrum Guide. https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf

Scrum.org. (2021). *The Nexus™ guide. The definitive guide to scaling Scrum with Nexus*. https://www.scrum.org/resources/nexus-guide

Sommerville, I. (2016). *Software engineering* (10th ed.). Pearson Education Limited.

Sun Microsystems. (n.d). *Core J2EE patterns: Data access object.* Oracle. https://www.oracle.com/java/technologies/dataaccessobject.html

Sutherland, J., & Coplien, J.O. (2019). *A Scrum book: The spirit of the game*. http://scrumbook.org/

Varanasi, B. (2019). *Introducing Maven: A build tool for today's Java developers*. Apress.

LIST OF TABLES AND FIGURES

IU Internationale Hochschule GmbH
IU International University of Applied Sciences
Juri-Gagarin-Ring 152
D-99084 Erfurt

Mailing Address
Albert-Proeller-Straße 15-19
D-86675 Buchdorf

media@iu.org
www.iu.org

Help & Contacts (FAQ)
On myCampus you can always find answers to questions concerning your studies.