Course Book

SOFTWARE ENGINEERING PRINCIPLES

IGIS01_E

iu
INTERNATIONAL
UNIVERSITY OF
APPLIED SCIENCES

SOFTWARE ENGINEERING PRINCIPLES

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

IGIS01_E
Version No.: 001-2023-1107
N.N.

TABLE OF CONTENTS

SOFTWARE ENGINEERING PRINCIPLES

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!

SUGGESTED READINGS

GENERAL SUGGESTIONS

Pohl, K., & Rupp., C. (2015). Requirements engineering (2nd ed.). Rocky Nook.

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

Sommerville, I. (2019). Engineering software products: An introduction to modern software engineering. Pearson.

Jacobson, I., Lawson, H., & Ng, P.-W. (2019). The essentials of modern software engineering. ACM Books.

UNIT 1

Tanenbaum, A. S. (2013). Structured computer organization (6th ed.). Pearson.

Chapters 2 & 3.1

UNIT 2

Gruhn, V., & von Brisinski, N. S. (2020). How to reduce risk effectively in fixed price software development. In 2020 IEEE/ACM 42nd international conference on software engineering: Software engineering in practice (pp. 132—141). Association for Computing Machinery.

UNIT 3

Kneuper, R. (2017). Sixty years of software development life cycle models. IEEE Annals of the History of Computing, 39(3), 41—54.

UNIT 4

Glinz, M., van Loenhoud, H., Staal, S., & Bühne, S. (2020). Handbook for the CPRE foundation level according to the IREB standard (pp. 1—139). International Requirements Engineering Board.

UNIT 5

Schmidt, R. F. (2013). Software engineering: Architecture-driven software development (pp. 43—54). Elsevier.

UNIT 6

Agutter, C. (2019). ITIL® foundation essentials—ITIL 4 edition: The ultimate revision guide. ITGP. Chapters 1, 2, 8, & 9

UNIT 7

Kruchten, P. (2003). The rational unified process: An introduction. Addison-Wesley.

Chapter 3

UNIT 8

Kruchten, P. (2003). The rational unified process: An introduction. Addison-Wesley.

Chapter 2

UNIT 9

Rubin, K. S. (2013). Essential Scrum. Addison-Wesley. Chapters 1 & 2

LEARNING OBJECTIVES

The aim of this course is to give students an insight into the technical and theoretical basics of **Software Engineering Principles**. This course will first cover Boolean algebra, enabling students to work on simple calculations. In addition to the general structure of computer systems, common challenges in the development of enterprise information systems are discussed. Students will be able to describe the structure of computer systems and communication networks, as well as different programming paradigms and their applications. The software life cycle is an important focus; students will learn how to distinguish between its phases, and how to describe the different process models of software development. Furthermore, the course book will cover the typical phases and activities in software engineering. These phases and activities will provide possible solutions and actions that can be used to address these risks.

UNIT 1

STRUCTURE AND ORGANIZATION OF INFORMATION SYSTEMS

STUDY GOALS

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

- explain what it means to encode information using binary code.
- name the units that make up a Von Neumann computer.
- identify the typical elements in distributed systems.
- name the elements of communication networks.
- understand the types of enterprise information systems.

1. STRUCTURE AND ORGANIZATION OF INFORMATION SYSTEMS

Introduction

To fully understand most of the principles of software engineering described hereafter, knowledge about the fundamental ways that information systems store and process information is required. Knowing the main components that comprise information system architectures is also essential. This unit will introduce the binary coding system and describe the basic components of contemporary computer system architecture. There will also be a discussion of typical elements of distributed systems and communication networks. The unit will close by giving insights into selected kinds of enterprise information systems.

1.1 0 and 1 as the Basis of All IT Systems

Bit
This is the smallest possible unit of information.

Digital systems store and process information in binary form, i.e., in the form of sequences of `0` and `1`. One **bit** corresponds to the amount of information in an answer to a question with only two possibilities, for example, “Is the sun shining?” Binary coding designates `0` or `1` to each of the two possible answers. `1` means, for example, “yes,” and `0` means “no.” Obviously, there are more than two possible answers to some questions. The coding can then no longer be completed in one bit; it will require bit sequences. These are used for storing information amounts that are greater than one bit.

The values `0` and `1` can be simulated relatively easily, both physically and using electrical engineering. The binary coding thus forms the transition from the world of digital computers to the real world. Some real world applications of binary are explained in the following:

- A light switch can hold two positions, “on” and “off,” so information up to one bit can be “stored” in a light switch.
- A simple flashlight can be switched on or off, so the light beam can be used to transmit information up to one bit.

Optical storage media, such as CDs, DVDs, and Blu-Ray discs, have a spiral track made up of small peaks and valleys. Every change (an increase or decrease) means `1`, and every non-change means `0`. This way, up to 25 gigabytes of data can be stored in one track on a Blu-Ray disc (single layer), and up to 128 gigabytes can be stored when using the BD-XL quad layer standard from 2010 (Risska, 2010).

- Magnetic storage media, such as hard drives, stores data on disks that are made of a magnetizable material and divided into many small areas. With a special write head, these areas can be set to one of two possible magnetic states. One state is spoken as "one" and the other as "zero."

All data stored in a computer system, as well as all programs that are executed on a computer system, manifest themselves as sequences of `0` and `1`. Only in this form can they be processed by the technical components, such as the central processing unit (CPU) or main memory.

Boolean Operations

The foundations of processing sequences consisting of `0` and `1` were laid in the nineteenth century by mathematician George Boole (The Editors of Encyclopaedia Britannica, n.d.). Today, the term Boolean algebra is used to describe an algebraic structure with the two elements, `0` and `1`, and the logical operations AND, NOT, and OR. The functionality of these operations can be implemented easily with electrotechnical assemblies and combined to form complex circuits. Today, all processing units are constructed on the basis of these simple operations. The table below gives an overview of the most important Boolean operations.

The Boolean operation "AND" is a two-digit operation. It evaluates two Boolean values, X and Y, to `1` if both values are `1`. Otherwise, AND evaluates to `0`. That is, if a lamp is connected to two light switches via an AND circuit, the lamp only lights up when both switches are in the "on" position. If a switch is in the "off" position, the lamp does not light up.

The Boolean operation "OR" is also a two-digit operation. It evaluates two Boolean values, X and Y, to `1` if at least one of the values is `1`. If a lamp is connected to two light switches via an OR circuit, the lamp lights up as soon as one of the two switches is in the "on" position. If both switches are in the "off" position, the lamp does not light up.

The operation "NOT" has only one input digit. It negates the value X to the other. The value `1` is evaluated as `0`, the value `0` as `1`. If a lamp is connected to a light switch via a NOT circuit, the lamp lights up when the switch is in the "off" position. The lamp does not light up when the switch is in the "on" position.

Table 1: Truth Table for the Boolean Operations AND, OR, and NOT

X	Y	AND (X, Y)	OR (X, Y)	NOT(X)
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Source: Created on behalf of IU (2023).

Storing Numbers and Letters

As described above, all digital data are stored as a bit sequence, but computer systems can also calculate ordinary numbers, and store and display letters and other characters. There are defined mappings and standards for this, for example, ASCII or UTF-8. In UTF-8, the sequence `01010011` is specified for the letter "S." In order to store letters and digits in binary, one only has to look up which sequences of `0` and `1` are defined for each. This information can be found in the UTF-8 table (FileFormat.Info, n.d.).

Table 2: Excerpt from the UTF-8 Encoding Table

Character	Binary encoding according to UTF-8
A	`01000001`
B	`01000010`
a	`01100001`
b	`01100010`
@	`01000000`

Source: Benner-Wickner (2021), based on FileFormat.Info (n.d.).

To perform mathematical calculations, the numbers from our conventional decimal system (with the digits `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`) are first converted into the binary system (using the digits `0`, `1`). The values are then calculated in the binary system and transferred back to the decimal system. The conversion is necessary because computer systems can only calculate with `0` and `1`.

1.2 Von Neumann Architecture

To fully comprehend the functionality of information systems, it is vital to know some key details about the basic composition of a computer system. This is especially true for performance issues in information systems.

von Neumann architecture
This is a general architectural concept for the construction of universal computers.

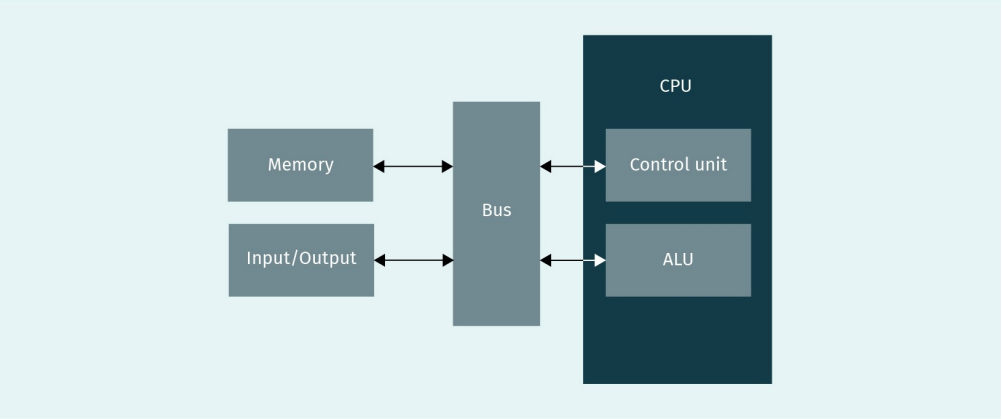
The basic functionality of all current computer systems was introduced in 1945 by John von Neumann (1903–1957) (The Centre of Computing History, n.d.) and is now called "**von Neumann architecture**." Before 1945, computer programs could not be called "software" because they were hardwired in, for example, electrical circuits or punch cards. Von Neumann developed the idea of storing the computer programs in a shared memory (the main memory) of the computer, together with the data being processed. Thus, the instructions for processing the data could now be easily changed, which made (re-)programming computers possible. The von Neumann architecture still serves as a conceptual template for the construction of universal computer systems. Almost every common computer system produced today follows the von Neumann architecture (The Centre of Computing History, n.d.). The spectrum of von Neumann computers ranges from huge mainframes to

laptops, game consoles, and smartphones. However, there is a technology emerging that breaks with this kind of architecture for the first time: the quantum computer. It can handle simultaneous operations using quantum phenomena, such as superposition. For now, this technology is not applicable for information systems discussed in this unit.

Units of the Von Neumann Architecture

The von Neuman architecture comprises five units: memory, control, input/output (I/O), the arithmetic logical unit (ALU), and a bus system connecting the units (Shipley & Jodis, 2003). In contemporary implementations of this architecture, most of the components—besides mass storage—are integrated into a central processing unit (CPU). Even graphical processing units (GPU) are integrated into modern processors. Each of the five basic units is described in detail as follows.

Figure 1: Von Neumann Architecture



Source: Created on behalf of IU (2023).

Memory

The memory of a von Neumann computer is used to store binary coded data and programs; there is only one type of memory. The available memory is divided into small areas that are numbered consecutively with a unique address. Data that are stored in a memory cell can be called up via their address. When accessing memory, the memory area in the storage medium is located and then read out or changed.

Control unit

The control unit and the ALU are essential elements of the computer's CPU. The control unit assumes the role of coordinator in the CPU. It is responsible for first loading the commands of a program to be executed from the memory into the CPU in the correct order and interpreting them. Then, the source and destination of the data to be processed must be interconnected with the ALU. Finally, the calculator must be informed as to which calculation it should carry out with the data.

Arithmetic logical unit

Like the control unit, the ALU is part of the CPU. It is the only component of the architecture that does calculations. The ALU has a number of executable arithmetic and logical functions. As previously mentioned, all operations in computer systems are carried out in the binary system. The available set of commands of the ALU is therefore mapped to simple binary operations that can be carried out very quickly (e.g., AND, OR, NOT).

Input output units

The I/O units are the interface between the system and its surrounding environment. They are responsible for the flow of incoming and outgoing data and programs. This includes communication with the user, for example, via screen, keyboard, and mouse, and communication with other systems via system interfaces.

Bus system

The bus is a data transmission system used by all units for communication. In a von Neumann computer, all data are transferred over the bus. Since both the provision of the data and the provision of the programs take place via the bus, the transmission capacity and the speed of the bus significantly determine the speed of the computer. Modern bus systems (e.g., PCI-Express for external devices or advanced micro devices [AMDs] HyperTransport for on-chip communication) work more like a point-to-point network than a bus system.

Unlike that which was common for computing systems at the time, the von Neumann architecture was not designed to solve a specific problem. With a von Neumann computer, all calculable problems can theoretically be solved because the internal structure of the computer does not determine how the data are processed. Since the programs are also in a changeable memory, the rules and instructions for how the data are processed can be adapted without having to exchange the hardware components of the computer system. The invention of the von Neumann computer has been an important paradigm shift, and has made programming software possible.

How the von Neumann Computer Works

In a von Neumann computer, the program commands are stored in successive memory cells within the memory. When a program is started, the control unit causes the first command to be loaded into the ALU. A command contains instructions for

- loading data from the memory into the ALU.
- carrying out calculations on data already loaded into the ALU.
- storing data.

After the command has been loaded into the calculator, it is executed. After the execution of the first command, the next instruction is loaded into the ALU by the control unit. This sequence is repeated until the program has ended.

Each command is executed within one step of the CPU clock. Today, the time spent adding two values within the ALU is less than 0.0000000001 s (1 nanosecond) (Leach et al., 2011). Imagine that, in this very small amount of time, even objects traveling at the speed of light can only move 0.3 m.

1.3 Distributed Systems and Communication Networks

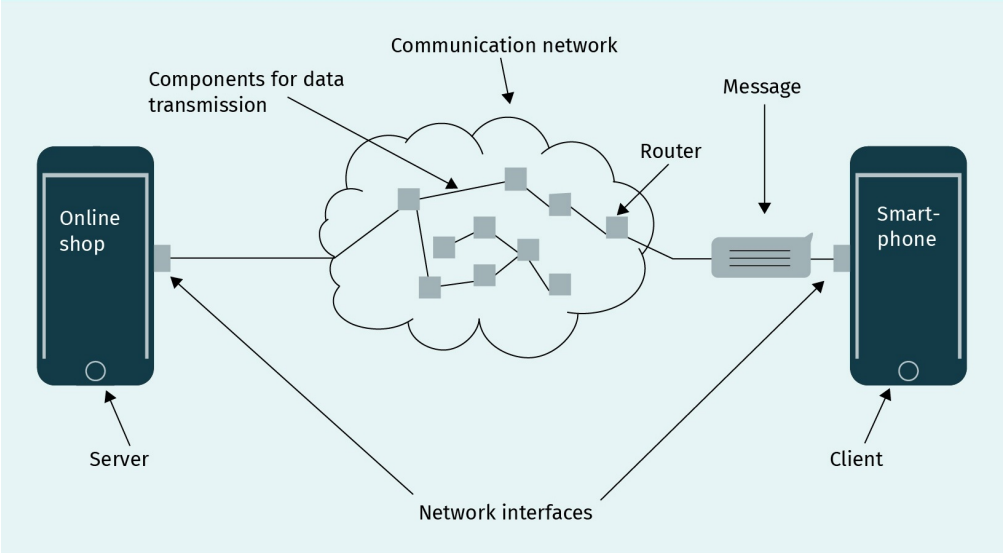
Another important evolution concerning the structure and organization of information systems is the switch from single mainframes to **distributed software systems**, which communicate autonomously using a (global) network infrastructure. Their underlying hardware and software components are located on computers that are connected by a communication network and communicate by exchanging messages via that network. For example, every online shop is a distributed software system: The customer's device is connected to the online retailer's computers via the internet. Searching for items, placing items in the shopping cart, ordering, and paying are examples of actions that are carried out via messages between the customer's computer and the online retailer's computer.

Distributed software system
This is a software system that can only be used in conjunction with several computers connected via a communication network.

Typical Elements in Distributed Systems

A distributed system consists of, at a minimum, a server, a client, a communication network, and an agreement on the structure of the messages to be sent. The figure below schematically shows the structure of a distributed system.

Figure 2: Illustration of a Distributed System



Source: Created on behalf of IU (2023).

Server

Servers are computers that make functions accessible to other computers via a communication network. The functions offered by servers are also called services. The following is a list of servers and their functions:

- Search engine servers enable information to be found on the internet.
- Weather service servers provide information on the current weather situation.
- Online retailers' servers offer search functions in the inventory.
- Email providers' servers enable the sending and receiving of emails.

Client

Computers using services are called clients, which are not to be confused with customers. If, for example, a browser is used to order something in an online shop, the computer on which the browser is installed takes on the role of the client. A network printer is a server that offers the printing services to the local network. Every computer that prints out a document via this printer is a client of the network printer.

Please note that both roles only relate to the offering and requesting of services, and not to the required hardware of the devices. Every laptop and smartphone can also become a server, i.e., a service provider, by installing and starting the appropriate software systems. Most systems within an enterprise network act as a server as well as a client, depending on the service considered. For example, the customer relationship management (CRM) system acts as a server, providing customer address data to the enterprise resource planning (ERP) system, which then becomes the client. The ERP system also acts as a server providing daily turnovers of every customer to the CRM system.

Communication network

A communication network connects clients and servers. It consists of

- physical components for data transmission, such as wires or radio antennas.
- technical components for routing and transferring data, such as routers, switches, repeaters, and bridges.
- network interfaces to connect a device to a communication network, such as an ethernet interface card or a Bluetooth chip.

Network interface

A network interface enables a computer to communicate with other computers. In addition to establishing the physical connection using a cable socket or a built-in radio chip (i.e., the hardware), support from the operating system (the software) is required. The components for data transmission ensure that information is transferred physically over a certain distance. The components of the data exchange ensure that the data are transported in a communication network using the optimal route from sender to recipient.

Message

Computers exchange information using messages. For example, the search term must be transmitted from the client to the search engine and the results from the search engine are sent back to the client. The structure and content of the message are application-specific and usually defined by the server. A message is defined as logically related information that is sent by an application. The network interface of a computer must transform the message into a bit sequence, which is then split for transmission into smaller data packets with a predetermined length. For example, a typical data packet transmitted through the internet via a digital subscriber line (DSL) connection can transport 1452 bytes of user data (Nozero, 2015). To send an image file with a size of 500 kilobytes from one computer to another computer, at least 353 data packets must be generated and sent over the communication network.

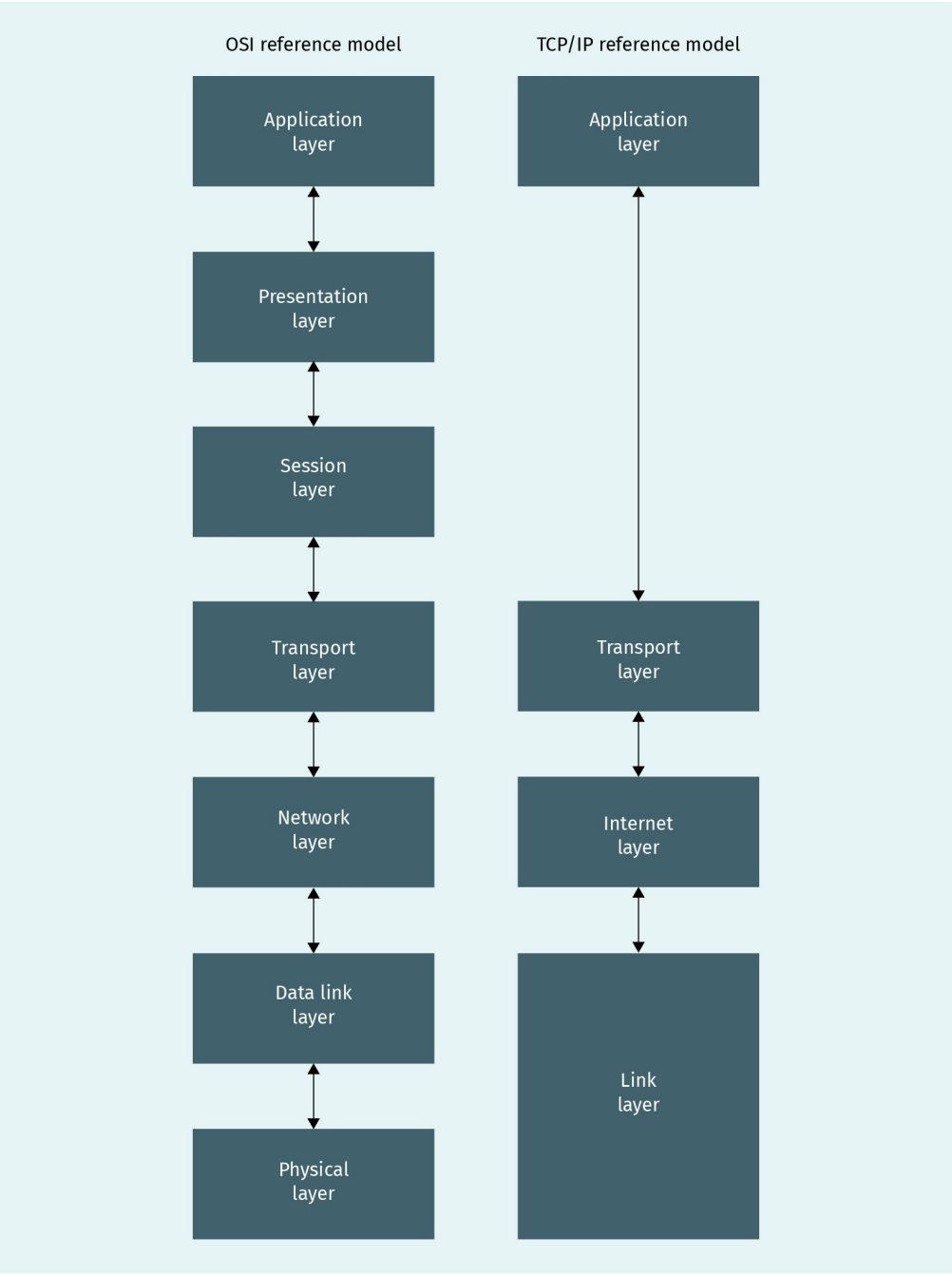
Conceptual Models for Communication

The messages must be transformed into data packets before they leave a computer. This transformation takes place in several steps. The type of message and communication network determine what happens in the individual steps. Both the internet protocol suite (TCP/IP) model and the open systems interconnection (OSI) reference model describe network architectures that encode the messages in a cascade of different layers in packets that are suitable for the transmission channel. They also decode received packets to combine them into a message. Please note that the OSI model has never been implemented fully in practice, but is still used to explain important steps in computer communication. The functions used for coding and decoding in a layer are called **protocol**. Each layer is responsible for a very specific aspect of the message exchange between computers.

Protocol
A set of conventions that govern the interaction of processes, devices, and other components within a system is a protocol.

The figure below illustrates the differences between the two reference models. The OSI reference model consists of seven different layers, while the TCP/IP protocol has four (in literature such as Tanenbaum [2013], TCP/IP has five layers because some authors add a physical layer). Thus, the OSI reference model is more detailed, and the functions described in the layers generally describe the architecture of networks. The TCP/IP reference model is essentially limited to the definitions in the transport and internet layers. Although it is less detailed, it is used almost everywhere.

Figure 3: Comparison of Conceptual Models



Source: Created on behalf of IU (2023).

Application layer

The functions (protocols) of the application layer in both reference models contain application-specific definitions for the structure and exchange of messages. For example, the Hypertext Transfer Protocol (HTTP) for calling up and transferring websites, and the Simple Mail Transfer Protocol (SMTP) for transferring emails are located.

Presentation and session layer

Elements of the presentation layer in the OSI model contain the definition of syntax and semantics of the information to be transmitted. They are responsible for tasks such as character encoding, data compression, and encryption and decryption of information. The session layer enables sessions to be set up between remote computers. With one session, the actions of both computers can be controlled and synchronized in a targeted manner, which enables smooth interaction, especially with long transfers or interruptions. Sessions are used, for example, when you log in to a web application or when you open a connection to a database.

Transport layer

In the transport layer, messages from the higher layers are divided into smaller units and passed on to the network or internet layer. In addition, the protocols use appropriate security functions in the transport layer to ensure that all data packets that have been sent arrive at the recipient. The transport layer of the sender communicates directly with the transport layer of the recipient, so there is a logical end-point-to-end-point connection in this layer, even if, from a technical point of view, there are many interconnected components for data transfer between sender and receiver. When the packets arrive in the wrong order—which can happen from time to time—the transport layer is capable of lining them up correctly using a sequence number.

Network layer or internet layer

The technical components for data transfer in networks (e.g., routers) work at the level of the network layer and the layers below. The network layer of the OSI model is responsible for the operation of the communication network. It takes care of the selection of the route through the network that a data packet travels, from the sender to the recipient. Depending on the geographical distance, there are several nodes on the path of a data packet from the sender to the recipient. In simplified terms, the network layer works like the post office: You can put letters into any mailbox, and, at some (end)point, letters will arrive at the correct destination address. The postal stations, distribution centers, and streets through which the letter was transported are not transparent for either the sender or the recipient.

The network layer also controls the quality of the transmission, which is useful if, for example, more data are sent on a route than the capacity of the transmission path allows. The internet layer in the TCP/IP reference model is very similar to the network layer in the OSI reference model. It is also responsible for the correct delivery of data packages and the avoidance of overload.

(Data) link layer

In the OSI reference model, the network layer is followed by the data link layer. With an appropriate protocol, it ensures that the data are transferred completely and without errors from one computer to the next in the route. In particular, the tasks of the data link layer are the flow control when the transmission speed is too high and error-handling due to physical transmission errors. In the TCP/IP model, this layer corresponds to the link layer.

Physical layer

According to the OSI model, the bit transmission layer is responsible for transmitting the binary coded digital data packets over the physical communication channel. This means mapping the bit sequences to be transported to the properties (electricity, light, and radio waves), transmitting them, and converting them back into bit sequences. This layer is not defined in the TCP/IP model, so all of the mechanisms described in the OSI physical layer are implicitly assumed to exist.

1.4 Enterprise Information Systems

Enterprise information system
This is a commercially used software system that is used to achieve business goals.

In the following, the term "**enterprise information system**" is used to refer to commercially used software systems and their system context. With the restriction to commercial use, the software systems that are used directly or indirectly to achieve business goals are placed in the foreground. The system context contains interfaces to the users of the system and technical interfaces to other systems.

Almost every medium and large organization depends on the use of information systems to achieve its goals. In particular, sectors with a fully digital value chain, such as banks, insurance companies, and media companies, depend on information technology (IT) support. In addition, every business model with the internet as a medium for marketing, sales, trade, or delivery needs information systems. This is also true for business models that include communication, networking, or data exchange via computer systems.

There are often attempts to classify enterprise information systems into categories. Unfortunately, there is no recognized, generally valid, and complete classification of systems. In this unit, we distinguish between four different kinds of systems:

1. Communication systems
2. Cross-sectional systems
3. Functional area information systems
4. Management information systems

Communication systems

Communication systems support interpersonal communication using email, video telephone, chat, or social networks. Microsoft Outlook, Zoom, and Slack are typical examples of this type of system.

Cross-sectional systems

All systems that are not communication systems but are also used as standard software are referred to as cross-sectional systems. Typical examples of cross-sectional systems are word processing, spreadsheets, and presentation systems, such as Word, Keynote, PowerPoint, or LibreOffice.

Both communication and cross-sectional systems are usually maintained by the software companies that developed them—essentially the "Big Five": Amazon, Apple, Google, Facebook, and Microsoft (Jones, 2019). These systems are used around the world, and are usually not customized to specific sectors or organizations. They are either standard products for which licenses must be acquired, or they are available as open source for download.

Functional area information systems

Functional area information systems (FAIS) specifically support business processes and activities in organizations. Some FAIS are industry-neutral applications, such as content management systems, accounting systems, and human resources management systems. A FAIS can also be an industry-specific solution; for example, for logistics companies, banks, or insurance companies. When no common, off-the-shelf system fits the organizational needs, FAIS are individually developed solutions. Of all the system classes, FAIS are most customized to the value-adding and supporting business processes. Enterprise Resource Planning (ERP) systems, such as those produced by System Analysis Program Development (SAP), can also be counted among the FAIS.

In order to support work steps and processes in organizations as efficiently as possible, FAIS are often the subject of software projects. The projects range from the organization-specific adaptation of industry-neutral and industry-specific solutions to the development of a completely new solution.

Management information systems

Management information systems are used to support planning and decision-making processes. Typical management information systems are data warehouse and business intelligence solutions. Purposefully cleaned and compressed data from the FAIS on procedures, processes, and business transactions are stored in these systems. These data can be summarized in the systems and prepared for the respective target group. In contrast to FAIS, the focus is not on business processes, but on data that are collected when the business processes are processed. As for FAIS, tools and solutions are available on the market for selected applications. However, since FAIS are often individualized, management information systems must also be customized for targeted use in the company as part of software projects.

SUMMARY

Before getting involved with complex enterprise software projects, we have to understand the very fundamentals of how information systems operate. Since information systems are digital systems, they store and process information as bit sequences (in the form of sequences of `0` and `1`). Because the values `0` and `1` can be simulated relatively easily, both physically and electronically, binary coding represents the transition to the world of digital computers into the real world. Boolean algebra with the operations AND, OR, and NOT enables the calculation with the two elements `1` and `0`. Today, all processors and chips are constructed on this basis.

The basic functionality of all of today's typical computer systems can be described with the von Neumann architecture. The core idea here is the common storage of computer programs and the data to be processed in a common memory of the computer. The architecture developed by von Neumann consists of five core elements: memory, control, input/output, the arithmetic logical unit (as part of the CPU), and the bus system.

Enterprise information systems are usually distributed software systems. They are located on computers that are connected by a communication network, and communicate by exchanging messages over that network. Typical elements of distributed systems are servers, clients, the communication network, and messages. The communication networks consist of the physical components for data transmission, the technical components for data transfer, and network interfaces to connect computers to a communication network. The architecture of communication networks is described with the help of reference models. The OSI reference model provides a complete and detailed description, but the TCP/IP reference model has become the underlying technical foundation for the internet.

The term enterprise information system refers to commercially used software systems and their system context. This summarizes systems that are used to achieve the goal of an organization and have interfaces to the users and technical interfaces to other systems. Information systems can be divided into the system classes' communication systems, cross-sectional systems, functional area information systems, and management information systems.

UNIT 2

RISKS AND CHALLENGES OF ENTERPRISE SOFTWARE ENGINEERING

STUDY GOALS

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

- identify the typical properties of enterprise software systems.
- understand the typical risks and problems that arise when carrying out software development projects.
- explain the causes of those risks and problems.
- identify which challenges of enterprise software projects can be derived from typical problems and causes.

2. RISKS AND CHALLENGES OF ENTERPRISE SOFTWARE ENGINEERING

Introduction

Software development projects in which enterprise information systems are to be created utilize processes that can basically be controlled with typical project management methods. However, compared to other manufacturing processes, there are relevant differences that result from the very specific properties of software systems. Therefore, this unit introduces the typical properties of software systems, explains the typical risks and problems that can be observed in software projects, and analyzes the causes that are primarily responsible for the problems. Building on this, the most important challenges in software engineering are presented.

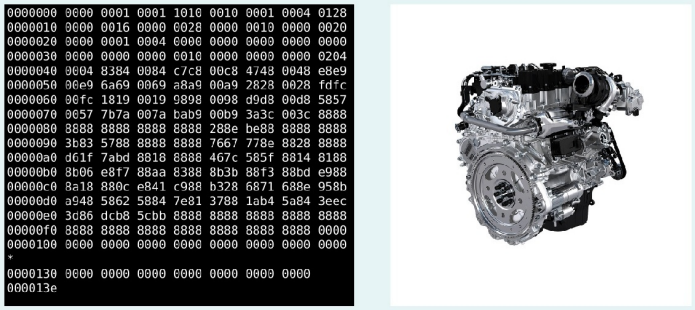
2.1 Properties of Enterprise Software Systems

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

- must support a wide range of functions.
- is part of a complex application landscape, and is connected to many other software systems via technical interfaces.
- is used by many users.
- should work on as many devices as possible under different operating systems.
- should be easily maintainable after it has been put into operation.
- is created by many people.
- consists of many different components and subsystems.

In contrast to industrial machines, for example, software is not tangible, but rather immaterial. A completed software system manifests itself as a very long series of `0` and `1` on a storage medium. This is another 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 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. In enterprise software engineering, complex, expensive, and business-critical systems are created that you can neither touch nor represent in a natural way (Sommerville, 2016, p. 18).

Figure 4: Software System versus Machine



Source: Created on behalf of IU (2023).

According to international standards, such as ISO/IEC 25010, enterprise software systems shall fulfill the following quality requirements (The International Organization for Standardization [ISO], 2011).

Quality Requirements

Functional suitability

When completely developed, software should match its (intended) specification. It is supposed to do exactly what was stated in the specification. If a system is inaccurately specified, whether the software created is correct cannot be determined.

One important aspect of functional suitability is interoperability. A software system is interoperable if it can be merged or integrated with other systems with little effort. In fact, a business application that works in isolation from other applications no longer exists. Cross-application interfaces (e.g., web services) are available in almost every company information system.

Reliability

The software is available and the user can use it in the context of their business activities. The “reliability” property is relevant depending on how critical the application of the software is and the height of the potential damage due to failure or malfunction.

Fault tolerance

If the software is not used as originally intended, or if a connected system does not behave as originally planned, a software system should behave tolerantly and not produce any uncontrolled errors or inconsistent data.

Usability

Software is usable when users find it easy to use. Since the success of Apple's products and the introduction of small, easy-to-use apps, usability has become increasingly important and often determines the success or failure of systems. A broader view on usability is given by the term "user experience," which also includes users' feelings when interacting with the software system.

Performance efficiency

Performance refers to how the system complies with requirements such as response time and resource consumption. Typically, the performance is either in distributed systems on which a large number of users work at the same time (e.g., Amazon, Facebook), or in systems that have to process large amounts of data at very specific times (e.g., banks, insurance companies). In short, this property can be checked by measuring, calculating, and simulating.

Maintainability

Enterprise software systems are often used over a period of several years, sometimes even decades. To be compliant with changing legal requirements and to react to changes in market or emerging technologies, new functions must be continuously implemented in existing systems. The ability of software to be changed in an economically viable way is known as maintainability. Maintainability depends heavily on the internal structure of the software and its documentation. In practice, maintainability decreases with increasing service life, so it makes more economic sense to completely replace or develop a system than continuing to maintain an existing system.

An important aspect of maintainability is reusability. It describes the probability with which a software system or parts of a system can be reused in a different context. Reusability must be carefully planned and considered when designing a software system architecture. It does not arise by chance.

Portability

The portability of software results from the effort required to make software executable on a different platform (e.g., database, operating system, mobile device) compared to the (new) development effort for software. Portability is a relevant property, especially for applications that are intended for the end user and therefore must work on as many device classes and operating systems as possible.

2.2 Software Engineering

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

The term **software engineering** describes the structured and methodical planning, creation, and evolution of software systems, as well as their operation. According to ISO/IEC/IEEE 24765:2017, software engineering is defined as the "application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software" (ISO, 2017, p. 8).

Smaller programs or apps for private use, or for highly specialized industrial applications, are sometimes created with little effort by one person. However, several dozen to a hundred people are often involved in implementing complex enterprise information systems over a period of several years. In order to implement and operate such systems reliably, software engineering must be applied. This means that such systems cannot simply be created according to the intuition of the project manager, but principles, methods, and tools must be used for this. Otherwise, software projects would be hopelessly lost in chaos.

Compared to other engineering disciplines, software engineering is a young discipline. Research into methods and principles with which the "software crisis" should be overcome was started in the 1960s (Cohane, 2017). At the same time, software systems became more complex, and they continue to do so. Supported by increasingly powerful hardware, more expressive programming languages, the networking of computer systems, the expansion of transmission capacities, the mobilization of computer systems, and resourceful development tools, newly created software systems are more powerful and more complex in their structure. Integrating artificial intelligence in software is a highlight of this development.

Because software systems are immaterial, their size and diversity are not limited by natural or physical laws. In contrast to other disciplines, such as architecture or electrical engineering, there are almost no natural restrictions (gravity, friction, or resistance). The power of software systems is therefore strongly limited by the ability to abstract—that is, the ability to develop abstract, logical structures—as well as the communication abilities of the people involved in the software project. Many recognized principles and methods of software engineering are therefore based on experience and knowledge in the development of complex information systems.

A core principle of software engineering is the division of the software project into different activities and the distribution of members of the software team into distinct roles, each with their own typical activities, responsibilities, and conflicting goals. Another core principle is the use of "patterns," i.e., the use of organizational and technical structures that have proven themselves in several projects.

Please note that the current state of research and technology in the field of software engineering is only a snapshot of a continuously evolving environment. Since there is no end in sight for the possibilities for further technological development, future technological developments will result in further tools and methods of software engineering that are not

known today. Therefore, software engineering research is concerned with the evolution of existing methods and tools, as well as with the development and evaluation of new methods.

2.3 Risks and Typical Problems

The complexity of enterprise software systems, together with the immateriality of software and the expectations of the client and user regarding the aforementioned properties of software, leads to a number of typical risks of software projects. In practice, you can find the following typical risks, among others, when working on software projects.

Project Termination Risks

The following is a list of risks that could lead to the termination of the project:

- Requirements turn out to be unrealizable in the course of the project. Typical causes for this 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 even 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

The following is a list of risks to the software applicability:

- During deployment of the software, it turns out that important business functions are missing or have been implemented incorrectly.
- 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 time window 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 even impossible to find.

Maintenance Risks

The following is a list of risks related to maintenance:

- The team responsible for maintenance cannot realize 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.

If you are interested in practical, cutting-edge discussions in the context of maintenance risks, please consider the proceedings of the International Conference on Technical Dept (TechDebt) published by The Association for Computing Machinery (ACM) (ACM, n.d.).

2.4 Root Cause Analysis

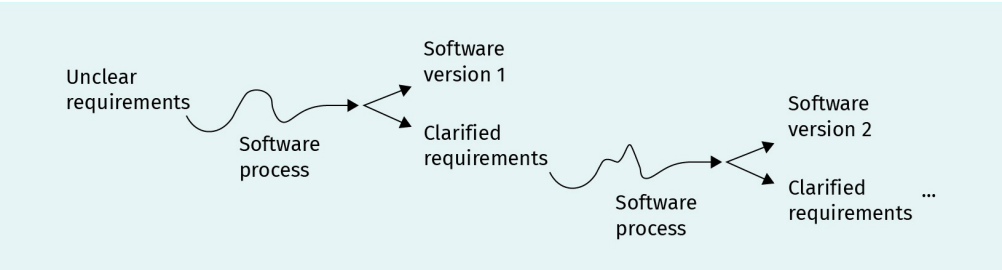
Compared to industrial processes producing mostly material goods, in which many work steps can be precisely planned in terms of duration and resource requirements, precise planning is usually not possible in software engineering projects. In particular, the exact costs, the specific scope of functions, and the technical design of a software system can only be precisely determined at the end of a project, which is, of course, too late.

The main influencing factors for software project risks are the complexity and immateriality of software systems. In addition, software development is a very knowledge-driven process. We will illustrate this using an example: Complex software projects are often started with an overarching business goal, e.g., “introduction of a self-care portal for customers of an insurance company.” 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. Participants will only learn these aspects as the project progresses.

A key feature of software engineering is that system requirements are only recognized in the course of the software process. Unfortunately, these requirements form the starting point for all further activities within a software project. As a rule, relevant requirements are only recognized by users and customers after they have seen a first version of the system.

A software engineer is now 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 created application does not implement the desired functions. However, at the beginning, users often cannot name needed functions. This is because they usually only recognize important and relevant functions when they test a first version of the system. The figure below schematically shows the interaction between recognized requirements and the software process.

Figure 5: Software Engineering is Strongly Knowledge-Driven



Source: Created on behalf of IU (2023).

The dilemma of being driven by knowledge is exacerbated in practice by the fact that there is usually not just one stakeholder who makes requirements for the system, but many stakeholders, often spread over several organizational units. This, in turn, means that many stakeholders may recognize additional requirements during the software process.

In addition to the stakeholders, new legal (compliance) requirements, technological developments, and the market situation can also have a significant influence on the requirements for software systems during a software project. This leads directly to a typical cause of failed software projects: changing and conflicting requirements.

In addition to commercial off-the-shelf software (COTS), such as word-processing software, there are also industry-specific software systems that are only relevant for very specific branches of industry, such as a specific integrated development environment (IDE) for mobile software development or a computer-aided design (CAD) tool for architects. Moreover, highly specialized custom software exists that is specifically designed and developed by individual companies for certain activities in the value chain. The share of this kind of software is steadily increasing due to the digital transformation of business processes. A lack of business knowledge on the part of software developers is a further cause of failed projects, especially for industry or company-specific software.

The development team must have a business understanding of the requirements formulated by the stakeholders so that it can deliver a usable software system. Because today's enterprise software systems are usually integrated into complex enterprise architecture 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.

If too much focus is placed on the technology used in software projects and the benefits for the user take a backseat, this is called technology-centricity. Software projects that focus on technology issues and problems also carry a high risk of project failure. It is often more exciting and convenient for developers to talk about technical topics and to concentrate on the use of the latest technologies than to actively deal with the users' technical

problems. As a result, a system is delivered that has possibly picked up on all new technology trends, but has been programmed beyond the actual needs of the user. Concepts such as human-centered design and design thinking address these issues.

Finally, the lack of communication or the lack of coordination among participants should be mentioned as a typical cause of failed projects. Different departments are usually involved in development projects for enterprise software systems, for example, marketing, sales, auditing, software development, operations, external consultants, the staff council, and the legal department. Each organizational unit has its own ideas and objectives with regard to the software system; there may already be differences within individual departments. This diversity is often supplemented by areas of responsibility that are not clearly defined and a lack of professional competence. In addition, most units use their own expert language. As a result, many terms are understood differently. Neglected communication management in such software projects quickly leads to different objectives, postponing or failing to make important decisions, and complication of the resolution of personal conflicts. This also increases the project risk.

2.5 Challenges in Software Engineering

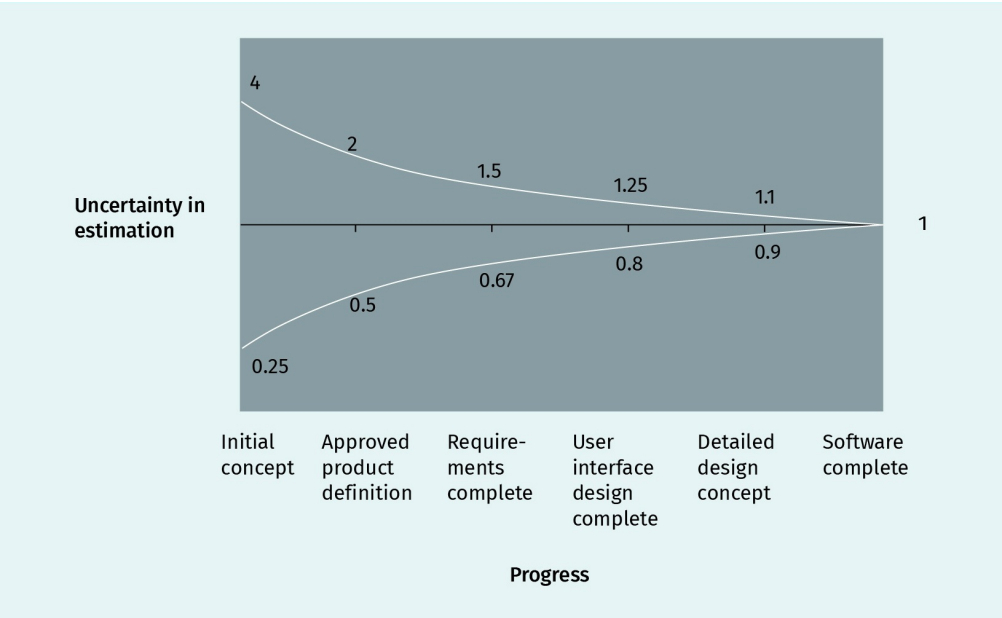
The aforementioned risks, problems, and causes result in a number of challenges that apply, in particular, to software development projects for enterprise information systems. In this section, we will discuss the following challenges in software engineering:

- economic uncertainty
- technological uncertainty
- communication
- conflicting goals
- complexity

Economic Uncertainty

A central challenge in enterprise software projects is dealing with and taking into account uncertainty throughout the course of the project. To illustrate the 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°. 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 process 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 precisely determine the level of effort. In contrast to production processes, in which the consumption of resources and the duration of the individual work steps can be calculated, a comparable precise forecast is usually not possible in software projects.

Figure 6: Cone of Uncertainty

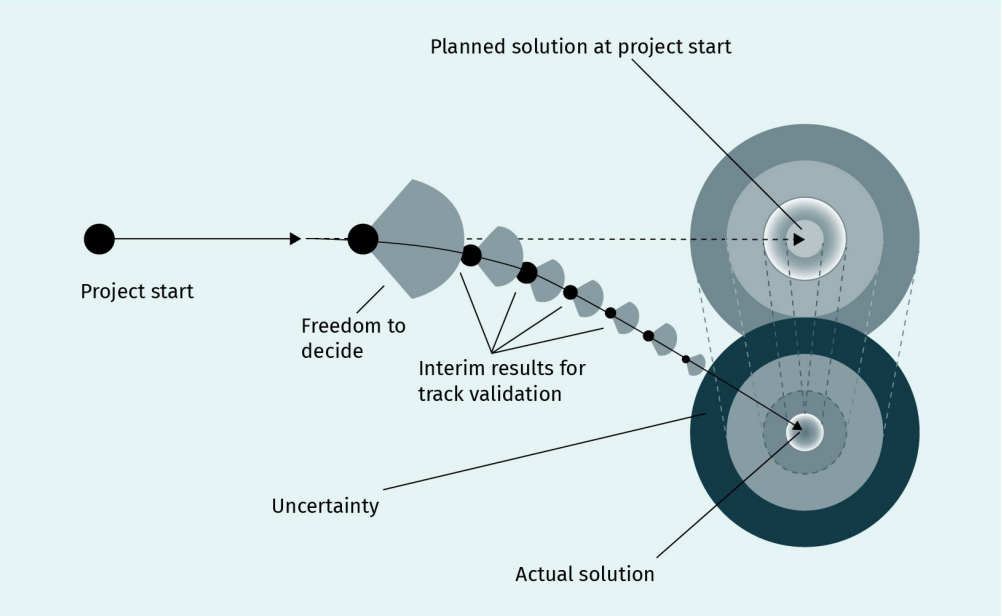


Source: Brückmann (2021), based on Bauman (1958).

Technological Uncertainty

Somewhat comparable to economic 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 of the project. The figure below schematically 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 on the basis of the current knowledge in order to support the requirements known at that point in time as well as possible. However, since requirements are only recognized 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 the technical solution becomes smaller as the project progresses, until a concrete solution has been worked out at the end of the project. This also means that, comparable to the economic uncertainty during the project, technical uncertainty must also be endured by the project team.

Figure 7: Technological Uncertainty



Source: Created on behalf of IU (2023).

Communication

The inclusion of various stakeholders, each with different individual backgrounds, knowledge, and interests, represents a further important challenge for enterprise software projects. Particular care must be taken here to establish and maintain willingness to cooperate. Furthermore, 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 quality assurance of artifacts generated in the software process.

Conflicting Goals

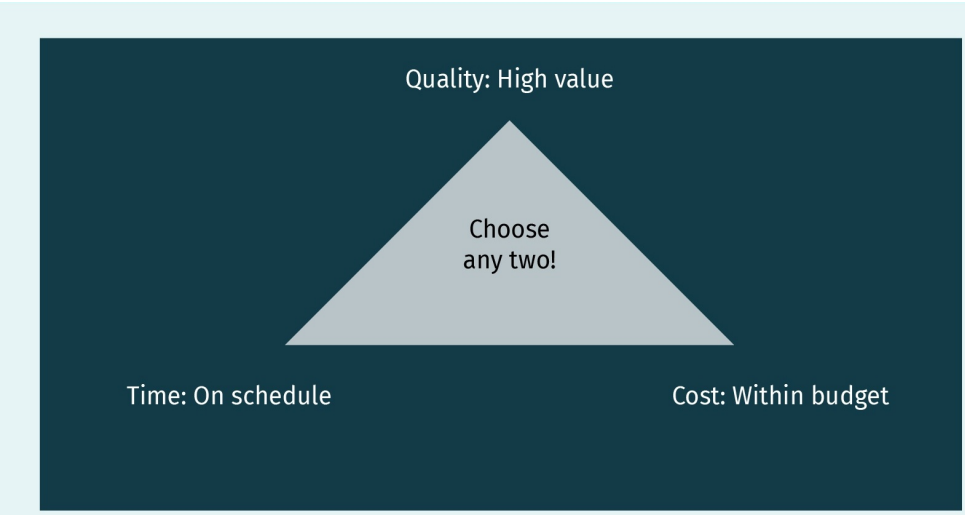
Even with software projects, customer satisfaction is the top priority, but decisions within software projects also range between quality, time, and costs. The figure below illustrates a typical decision dilemma with the help of the **magic triangle**: If you improve one target parameter, this is usually at the cost of the other two target parameters.

Magic triangle
The magic triangle illustrates competing goals.

For example, if quality is very important to a project, quality assurance activities are at the expense of timely completion and budget. If the focus is on project costs and all 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 will not be kept.

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

Figure 8: The Magic Triangle: Quality, Time, and Cost



Source: Created on behalf of IU (2023).

Complexity

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

The use of software models is a key practice in software engineering to which there are currently no sensible alternatives. Various types of models have been developed to master complexity, which illustrate static and dynamic aspects of software systems in varying degrees of detail. Under the premise of uncertainty and knowledge gain during a software project, 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.

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 with all potential eventualities. Either an exaggeratedly detailed project plan is drawn up, which is intended to predict, for the entire project, which activities are to be carried out by whom and with what effort. The business requirements and solutions of the software system are analyzed on a very 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. The attempt is made to hide the uncertainty by producing complex analysis results, for example, in the form of large and complex software models. The team often does not just try something, but first analyzes

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

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. However, the analysis models that were elaborately created at the beginning often become obsolete due to newly gained knowledge during the implementation and presentation of the first software version.

SUMMARY

In addition to their complexity, a typical property of enterprise software systems is, in particular, their immateriality. Software is neither visible nor tangible. Of course, you can still review the code, but imagine that even the software running in a ten-year-old car contains close to 100 million lines of code. In addition to supporting business functions, software systems also require the following properties: functional suitability, reliability, fault tolerance, usability, performance efficiency, maintainability, and portability.

Software engineering describes the structured and methodical planning, creation, and evolution of software systems and their operation. A core principle of software engineering is the division of the software project into various activities, the assignment of team members to responsibilities for certain activities, and the use of organizational and technical patterns that have proven themselves over several projects.

The complexity of enterprise software systems, together with the immateriality of software and the expectations of the client and user with regard to the aforementioned properties of software, lead to a number of typical risks in software projects. One factor influencing project risks is that software development is a highly (domain-)knowledge-driven process, and the requirements for the system in development can change in the course of a project. A lack of business knowledge in the development team or a lack of communication and coordination are typical causes of failed software projects.

Derived from typical risks and their causes, the following fundamental challenges are found in enterprise software engineering: dealing with economic and technological uncertainty during the project, communication between those involved in the project, dealing with conflicting goals, and the technological complexity of enterprise software systems.

UNIT 3

SOFTWARE LIFE CYCLE: FROM PLANNING TO REPLACEMENT

STUDY GOALS

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

- explain the phases of the software life cycle.
- identify the characteristics of each phase.
- understand the activities within the life cycle phases.

3. SOFTWARE LIFE CYCLE: FROM PLANNING TO REPLACEMENT

Introduction

Software systems go through a life cycle, which consists of different phases. Each phase is characterized by typical activities, people involved, risks, and challenges. In this unit, we will examine the details of each phase. Beginning with the planning phase, we will show how basic customer needs are determined. The development phase continues, applying all of the constructive activities required, from the initial order to the deployment and implementation of the system. After the development work has been completed, the software system must be deployed to the operational environment and integrated into the existing application landscape within the operation phase. The maintenance phases include all activities concerning the management of problems and incidents, as well as the functional evolution of the operational application. At the end of the software life cycle, the application is taken out of service.

3.1 The Software Life Cycle at a Glance

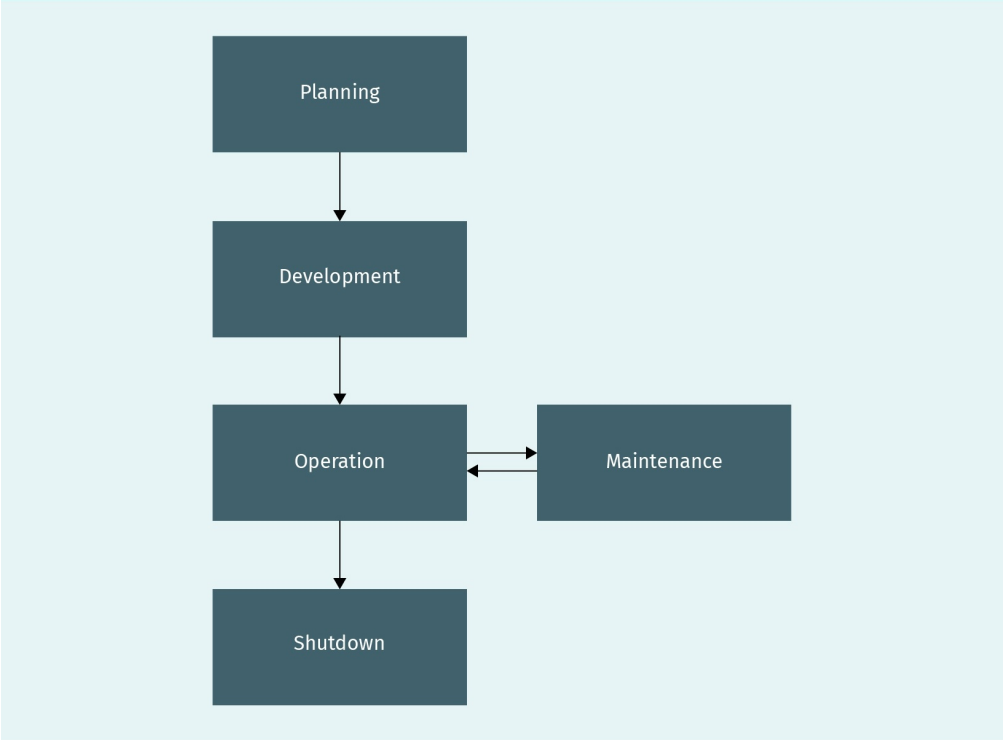
Software life cycle
This is a software system or software product cycle initiated by a user need or a perceived customer need, and terminated by discontinued use of the product or when the software is no longer available for use.

As shown in the figure below, the **software life cycle** can be roughly divided into five individual phases (The International Organization for Standardization, & The Institute of Electrical Engineers, Inc., 2017):

1. Planning, which includes the activities before the start of a software project
2. Development, which includes the activities between the start of the project and the deployment of the finished system
3. Operation, which is the start-up and operation of the system in the target environment
4. Maintenance, which is the maintenance and evolution after deployment
5. Shutdown, which includes the activities used to take the system out of operation

The software life cycle model and its phases are used for medium- and long-term project planning. Hundreds of applications are often used in large companies. Based on the current phase in the life cycle, information technology (IT) management can get an overview of the current status of its system landscape and draw up personnel, resource, and investment planning. Furthermore, based on the determination of the current phase in the life cycle, certain activities and methods of software engineering can be derived.

Figure 9: Software Life Cycle



Source: Created on behalf of IU (2023).

The life cycle is run through in order, from planning to shutdown. In practice, the phases of the life cycle cannot always be clearly separated, but flow into one another. The operation and maintenance phases can be run several times in succession before the system is switched off.

3.2 Planning

The initial phase of a software life cycle is planning. In this phase, all activities that are carried out before the start of software development activities can be determined. This phase is initiated by IT management, who are actively involved in the planning phase, and make the decisions relevant to the structure of the project. Starting with the determination of the requirements, through to the financial planning to the procurement, the conditions for all further phases of the project are identified and determined.

Determination of Needs

The reason for introducing a software system is the realization that there is a need for a system. The specific reasons for the need for a new software system are very diverse and depend on the area of application of the IT organization. Typical occasions include

- the replacement of existing legacy systems.
- business demands.
- technological evolution.

Replacement of existing legacy systems

IT systems are subject to aging and therefore have a limited lifespan. After the initial commissioning, software systems evolve, and they are usually revised and reworked several times, either to correct software errors or add functions to the system. Depending on the lifetime and the frequency of adjustment, the maintenance costs are disproportional to the benefit achieved, so it makes more sense to replace an existing system with a new system. In addition to the technical aging process, the know-how for the maintenance of existing systems is often unavailable, for example, because the technology used is no longer supported by the manufacturer, or the developers have left the company.

Business demands

Enterprise software systems play a key role in value-adding processes (particularly in companies in the insurance, media, and logistics sectors). Companies in these industries develop competitive advantages using highly specialized software systems. For this reason, corporate IT must react quickly to new business requirements and support the business departments with suitable systems in a targeted manner. If business requirements cannot be covered by existing systems, new software systems must be provided.

Technological evolution

Continuous technological evolution enables new fields of application and business that must be supported by appropriate IT solutions. Therefore, changing technical constraints also lead to a need for software systems. For example, it was only with the spread of web browsers that the large-scale provision of web applications became possible and thus, location-independent work. Likewise, the spread of smartphones is creating a need for a new class of application, the "apps."

Make-or-Buy Decision

After the need for a new software system has been identified, the following questions must be answered:

- Are software solutions already available that cover the need (e.g., content management systems, e-mail systems, word processing)?
- Are there standard products that can be customized to company-specific requirements (e.g., SAP Business Suite, Oracle E-Business Suite, IBM WebSphere)?
- Does new software need to be developed?

Make-or-buy decision
This is deciding whether an existing system is to be procured or a new system is to be developed.

This is called a **make-or-buy decision**. Depending on how relevant the required software is for the company's business activities and how serious the possible consequences of a wrong decision are, preliminary studies, test installations, or initial application prototypes are created and tried out as part of a make-or-buy decision.

Depending on the result of the make-or-buy decision, more specific time and resource planning takes place. A first project plan is defined up to the introduction, and the required resources are determined and planned.

Procurement

If a new software system is to be created or a standard solution is to be customized, a corresponding order must be placed. Whether the order will be carried out by an external service provider or an internal IT department must be decided. If it is a public IT organization, usually, the contract must be advertised publicly according to each country's law.

In the case of public tenders for the development by external service providers, typical activities from the "development" phase must be carried out in the "planning" phase when the requirements for the new system are determined. To bring about a decision during hand-over as to whether the service provider fulfills all contractual obligations, the results of the requirements determination must already be documented when the contract is awarded.

3.3 Development

When the decision has been made that a new system should be created or a standard solution should be customized, the development phase follows. This phase is where all of the constructive activities required from the order to the deployment of the system take place, and the system is implemented. The activities are also summarized under the terms "application development" and "software development." During the development phase, more people work on the system at the same time than during any other phase. The software development activities described below are the core activities of software engineering.

Requirements Engineering and Specification

Based on the broad business requirements determined in the planning, business requirements are further detailed and refined. This is called requirements engineering. All relevant stakeholders of the system are involved. After the business requirements have been determined, the required properties of the system in development are documented by a specification.

Software Architecture and Implementation

The internal structure of the software system (the software architecture) is determined based on the specification. For this purpose, the needs of the stakeholders are analyzed and weighed against each other, and an architecture definition is developed through several decisions and design activities. The program code of the system is implemented according to these specifications, and the system is created accordingly. The source code of a system is programmed during implementation, and the system is thus constructed.

Quality Assurance

All artifacts generated during creation (e.g., software architecture, source code, and test cases) are checked to determine whether they meet specified requirements. Measures and activities for quality assurance are already being carried out during all activities. However, the system can only be tested completely after implementation and integration has been finished. The program code is therefore tested in different test stages, depending on the status of the project.

IT organizations often have their own departments (e.g., application development) that are responsible for creating software. After the software has been implemented and tested, it is handed over to the department responsible for operating the system.

3.4 Operation

After the development work has been completed, the software system must be deployed in the operational environment and integrated into the existing application landscape. The security and availability of the system must then be guaranteed so that all users can work productively with the system as expected. This phase in the software life cycle is called “operation.”

Operational Environment Provisioning

During the development phase, many people are busy designing the system at the same time, and the system is not yet involved in a company’s value-adding activities. In the operation phase, this relationship is reversed: Work on the program code is stopped and the system is used intensively by its users. In large companies, systems are used by several thousand users at the same time. If customers or end users are involved (e.g., Amazon, Netflix, Facebook), the system can be operated by as few as ten users, or over one hundred thousand users at the same time.

This means that the security risk also increases. The more people that have access to a system via the internet, the greater the risk of system attacks or fraud. In addition, no data should be lost in the event of a hardware failure or the occurrence of a software error. At the same time, the system should be able to cope with load peaks if necessary, but should only use as many resources as it truly needs when operating under normal requirements.

The aim in operation is to ensure the application’s quality requirements, such as security, availability, and scalability. This means that the department responsible for operation must provide the appropriate infrastructure, for example, in a data center. Typical elements of this infrastructure are special hardware systems (rack servers), software systems (operating systems, monitoring systems, and virtualization systems), networks (ethernet, WiFi, and mobile), storage systems (databases and backup systems), and security systems (firewall, virus scanner, and cryptography systems). This also includes securing the power supply, air conditioning, and the connection to external networks (internet), as well as user administration and allocation of access rights.

In contrast to development, which aims to quickly add functions to software systems, long-term stability must be ensured during operation. This results in a conflict of objectives between development and operation. On the one hand, there must be a quick reaction to business requirements, but on the other hand, the system should run reliably and safely. Current interdisciplinary approaches, such as development operation (DevOps), address these issues.

Integration

After development, enterprise software systems must be deployed in the operational environment. In addition to being installed in that environment, the system must be integrated into the application landscape. This means that it must be connected to existing systems using the technical interfaces provided by the developers. For example, an insurance company's existing system must be connected to the collection and disbursement system so that premium invoices can be sent automatically. Another example is a system for fund management that must be connected to a service providing current exchange and stock exchange rates. CRM systems have to be connected to a system for the automatic validation of addresses.

The new system is only connected to other systems in the operational environment after all tests have been completed, so that real data in operational systems are not accidentally changed during development

Provisioning

After the integration is complete, a new system can be put into operation. At this point, all technical interfaces of existing systems must be converted to the new system. For web applications, for example, it must be ensured that entering the uniform resource locator (URL) allows the user to access the new application.

To ensure availability and security, the new application must also be connected to technical monitoring systems that continuously provide management with information about resource consumption (central processing unit [CPU], main memory, data storage, and network access). In addition, IT security must include the new application in the list of monitored applications.

3.5 Maintenance

When software systems are provisioned for the first time, the development phase has already been concluded. Nevertheless, change requests by the user must also be implemented for applications during operation. Necessary changes may include correcting spelling errors in the application interface, importing version and security updates, or adding completely new functions. This phase in the software life cycle is called "**maintenance**." It includes all activities concerning the management of problems and incidents, as well as the functional evolution of the operational application.

Maintenance
This is the totality of activities required to provide cost-effective support to a software system.

Upkeep versus Evolution

Maintenance includes the upkeep and corrective adjustment of software systems, which mainly involves the elimination of detected errors. Enterprise information systems are not perfect when they are delivered. Despite the quality assurance measures, it is to be expected that there will be residual errors in the system. In addition to eliminating errors, adjustments with the aim of better runtime behavior or more economical use of resources are also maintenance tasks.

Furthermore, maintenance incorporates the functional evolution of the system. If a software system is to be expanded with functions, or if functions have to be changed, these are activities for the future development of the system. In the case of a planned system lifetime of ten years or longer, future developments in the design of the system must be considered when creating software.

Cross-Cutting Activities

Both upkeep and evolution mean that the program code of a system that is already in operation must be changed. This means that activities of the development phase are carried out in the maintenance phase. The adapted system must then be put into operation, so activities of the operation phase must also be carried out. After the initial provisioning of a software system, the phases of maintenance and operation alternate in relatively short cycles.

In the release plans, dates are given for each application at which an update of the application will be made available. These plans help with the cooperation of business departments (which have change requests), application development (which implement the change requests), and IT operations (which bring the changed application into productive operation).

For software systems, it can be observed that the effort for maintenance over the entire life cycle approximately doubles the effort for creation. With a development effort of 500,000 euros for an application, maintenance costs of one million euros or more can be expected. The actual effort differs greatly from application to application. For periods of 15 years or more, the maintenance costs can be significantly higher than the development.

3.6 Shutdown

At the end of the software life cycle, the application is taken out of service. This final phase in the life cycle is called shutdown or disposal (The International Organization for Standardization, 2017). As part of maintaining the application landscape, IT management decides whether and when an application is to be shut down. Typical reasons for a shutdown include

- the replacement of existing legacy systems with new systems.

- the consolidation or standardization of applications when merging IT organizations.
- giving up business areas or outsourcing activities from the company.

Similar to the provisioning, the system and its technical and organizational environment must be taken into account during the shutdown.

Removal

Since systems within an application landscape are connected to one another via technical interfaces, all dependencies of the application being switched off must be identified and resolved. Every actively used technical interface of the old system must either be covered by a new or existing alternative system. Under certain circumstances, each system connected to the legacy system must be adapted.

Particularly in the case of very old systems with obsolete technologies and outdated documentation, the extraction is a critical activity. Technical dependencies must be identified, and the system behavior at the interfaces must be reproduced in the event of a system replacement. The basis for this is usually the system documentation. However, if the documentation is incomplete or outdated, dependencies can only be identified by reverse engineering or testing the system context while the legacy system is detached.

If there is no current documentation on the implementation of the function within the legacy system, the program code must be analyzed and the behavior of the new system must be specified with the results. However, this requires expert knowledge of the technologies used in the legacy system. Unfortunately, there is a high risk that those knowledgable about the old technologies have already left the company.

Data Migration

If data are stored in the legacy system, they might be migrated as part of the shutdown. This means that the data in the old system must be transferred to another system for further use. Unfortunately, systems are getting very old, especially when they are used for the storage and management of company-critical data, for example, the inventory system of life insurance companies or the system for managing bank accounts. One challenge when migrating data is finding an appropriate mapping of the old data schema to the new data schema. Another challenge is the amount of data to be migrated. Even when transferring several million contract data—a number quite common for health or life insurance companies—there must be no mistake. Sometimes it is more economical for the company to dissolve old contracts or to have customers switch to a new contract version by giving them a discount than to migrate the data from the old system to a new system (internet providers, for example, do this).

Contract Termination

In order to operate a software system, an appropriate operational environment must be provided and maintained. The infrastructure required for the legacy system must be analyzed to determine whether term-based licenses and usage rights have been acquired. There are often application- and user-dependent license costs for the operation of certain

hardware or operating systems. Corresponding contracts must be terminated, or they may not be automatically extended. Maintenance contracts with external service providers must also be taken into account.

Retraining

As part of the disposal, IT management must also ensure that employees responsible for the operation and maintenance of legacy applications find other tasks. In particular, if the shutdown leads to a technological change and the IT infrastructure is converted from proprietary mainframes to standardized systems, or the programming language is changed from Cobol to Java or .NET, programs for retraining employees must be designed and implemented in advance.

SUMMARY

Enterprise software systems go through a life cycle, which can be divided into the following phases: planning, development, operation, maintenance, and shutdown. The software life cycle model and its phases are used for medium- and long-term project planning and application portfolio management. In the initial "planning" phase (starting from the determination of broad requirements, through make-or-buy decisions and resource planning, right up to the procurement), the constraints for all further phases in the course of the project are determined.

In the development phase, all constructive activities take place. These activities are also called application development or software development. They include the core topics of software engineering: requirements engineering, specification, architecture, implementation (i.e., programming), and quality assurance.

After the development work has been completed, the operation phase follows by installing the system on the operational environment, integrating it into the existing system landscape and putting it into operation (i.e., switching it on). The security, availability, and scalability of the overall system (including its technical infrastructure and hardware) must be ensured during operation.

The maintenance phase includes all activities related to error correction (upkeep) and evolution of the application after it has been provisioned. Whenever the program code of an operational system has to be changed, maintenance work includes activities of the development phase and the operation phase. Maintenance can be coordinated with the help of release plans, and maintenance costs are known to be at least double the development costs.

The final phase in the life cycle is shutdown. In this phase, all preparations are made to dispose of an application. To remove the system from the application landscape and eventually migrate its data to the new system, existing license and maintenance contracts must be terminated and, if necessary, personnel must be retrained for new tasks.

UNIT 4

REQUIREMENTS ENGINEERING AND SPECIFICATION

STUDY GOALS

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

- explain what requirements engineering is and who is involved.
- identify the activities that are carried out in requirements engineering and explain how they are related.
- understand what a specification is and how it is used in software engineering.
- define the system properties that are described in a specification.

4. REQUIREMENTS ENGINEERING AND SPECIFICATION

Introduction

Before the construction of the software can begin in a software project, the end goal of the system must be determined. Based on that goal, the system functions must be described in detail so that the development team can create and implement a suitable architecture. The relevant steps for determining, documenting, and coordinating these functions from a business view are known as requirements engineering (RE). Based on that business view, system requirements can be broken down into an assignable contract by writing a specification.

4.1 Requirements Engineering

A rough definition of the goal already takes place in the "planning" phase of the software life cycle. Based on this goal, all functions and properties of the system must be determined gradually. The system should support this after the development has been completed. An example of this is as follows: In the planning phase, it is decided to integrate a system with which the application for and billing of business trips can be digitized in order to completely dispense with paper forms. Based on this general goal of the system, the specific functions must now be determined, documented, and coordinated so that the software development team can implement them.

Requirements
These are the functions and properties that a system should support or provide.

In software engineering, the necessary functions and properties are called **requirements**. The activities for determination, documentation, negotiation, and administration are summarized under the term "requirements engineering." According to the *Certified Professional for Requirements Engineering (CPRE) Glossary*, requirements engineering (RE) has the following goals (Glinz, 2017, p. 18):

1. "Knowing the relevant requirements, achieving a consensus [...] documenting them according to given standards, [...] managing them systematically
2. Understanding and documenting the stakeholders' desires and needs
3. Specifying and managing requirements to minimize the risk of delivering a system that does not meet [them]."

The following characteristics for requirements engineering can be derived from this definition: Several people, or groups of people, are always involved in RE activities and must cooperate with one another. Specifically, these are people or groups of people whose area of work is either influenced by the system to be created, who interact with the system, or who are involved in the creation process of the system. All affected or involved people are called "stakeholders."

Requirements engineering is not a single, completed activity at the beginning of a software project. RE takes place in several cycles, or iterations. By carrying out RE activities several times, requirements for the system are determined and refined. Please note that, since software engineering is a highly knowledge-driven process, RE is usually carried out alongside the project, not only at the beginning. According to the *CPRE Handbook*, RE comprises four core activities (Glinz et al., 2020):

1. Elicitation
2. Documentation
3. Validation
4. Management of requirements

No fixed sequence can be described for these activities because a responsible requirements engineer decides what type of activity will take place next, depending on the current project situation.

Requirements Elicitation

The aim of requirements elicitation is to identify the system requirements that are needed to achieve the goal. To do this, the requirements must be recognized and understood in the level of detail required for the current project situation. Please note that requirements elicitation is also intensive in early phases of contemporary approaches, similar to design thinking. Below, you will find an example of a typical project situation where requirements elicitation occurs. The example is followed by a more detailed step-by-step approach for determining requirements.

Example

At the start of the project, it is important to determine all relevant basic functional areas without going into details (e.g., “creating a business trip,” “editing a business trip,” “creating a travel expense report,” “reimbursement of expenses,” “approving the reimbursement of expenses”). The more advanced a project is, the more detail is needed about the individual requirements. For example, for the functional area “editing a business trip,” it must be specified which input fields are available, which fields are mandatory, which conditions must be observed, and who is allowed to make changes. The systematic elicitation of requirements is divided into the following four steps:

1. Determine the system context. It must be clear which stakeholders and other systems are directly dependent on the creating system. These must be explicitly considered in RE.
2. Determine sources for requirements. Typical sources for requirements are stakeholders, documents (e.g., laws, guidelines), and other systems (e.g., legacy systems to be replaced or competing systems).

3. Select suitable elicitation techniques. Depending on the requirement source, project situation, and type of requirements, a suitable elicitation technique or a combination of different elicitation techniques must be selected (e.g., distinct techniques for gathering, questioning, observation, collaboration, or creativity [Glinz et al., 2020]).
4. Apply techniques. Requirements are determined from the specific sources with the selected elicitation techniques in the degree of detail required for the current situation.

Requirements Documentation

The aim of requirements documentation is to ensure that the current state of knowledge is secure for all stakeholders, and that everyone involved can obtain an overview at any time. In cross-organizational software development, requirement documents are a part of legally valid contracts. Based on such contracts, it is later checked whether the agreed service was provided by the software manufacturer.

To document complex structures and relationships in a suitable manner, graphic models are often used in software engineering, in addition to the description of requirements in text form (software models). The specific form of documentation, however, always depends on the type of requirements that are to be documented, the purpose of the documentation, the group of people for whom the documentation is intended, and regulations or agreements in the documentation format that apply to the project. Four steps can be identified for the systematic documentation of requirements:

1. Determine the purpose and target group of the documentation. The documentation of requirements supports the communication process in the software project. Therefore, before the documentation, the purpose and target group for which the documentation is created must be determined.
2. Select the level of detail and type of model. The level of detail and type of model (e.g., text, graphics, software models, and prototypes) are determined depending on the purpose and target group.
3. Document requirements. The determined requirements must be documented in a form suitable for the purpose and target group.
4. Check the documentation. After completing the documentation, it must be decided whether the type and form still fit the purpose and the target group. This is used to check, once again, whether the documentation team has gotten something wrong, or whether the target group or purpose have changed during the documentation.

Requirements Validation

Since all further activities in software engineering directly depend on the determined and documented requirements, the requirements must be checked before being released for implementation and coordinated with the relevant stakeholders. The aim of the validation is to ensure the quality of the amount of documented requirement regarding content, documentation, and negotiation. It must be ensured that the requirements have high quality documentation so that misunderstandings due to ambiguities are avoided and contradicting or competing requirements are identified.

If conflicts or contradictions are identified during an audit, these must be resolved with the involvement of the stakeholders. Many different stakeholders must be involved, especially in the creation of enterprise information systems. During requirements engineering, therefore, competing or mutually contradicting requirements are usually determined and documented. Conflict management plays an important role in the validation process. Like the core activities of elicitation and documentation, validation can also be divided into four steps:

1. Define validation criteria. This determines the focus of the check, which is particularly necessary when validating extensive documents to adhere to the specified schedule.
2. Select test principles and test techniques. Depending on the validation criteria, the time available and the status of the documentation, validation principles and validation techniques that consider the criteria are selected (e.g., review, walkthrough, and creation of software artifacts).
3. Perform validation and document results. Results should be documented during the validation. The error correction is carried out afterwards.
4. Negotiation of requirements and conflict management. If conflicts or contradictions are identified during the examination, negotiation with the relevant stakeholders must be carried out, which may be followed by conflict resolution techniques.

In addition to the core activities presented above, the *CPRE Handbook* also discusses the management of requirements as a core activity (Glinz et al., 2020).

4.2 Specification

The aim of the specification activities is to create technical documentation of the externally relevant requirements, according to which, a software system will be produced. Based on the knowledge gained through RE, in which business requirements were determined, documented, checked, and negotiated, a technical documentation of the system is created by writing a specification.

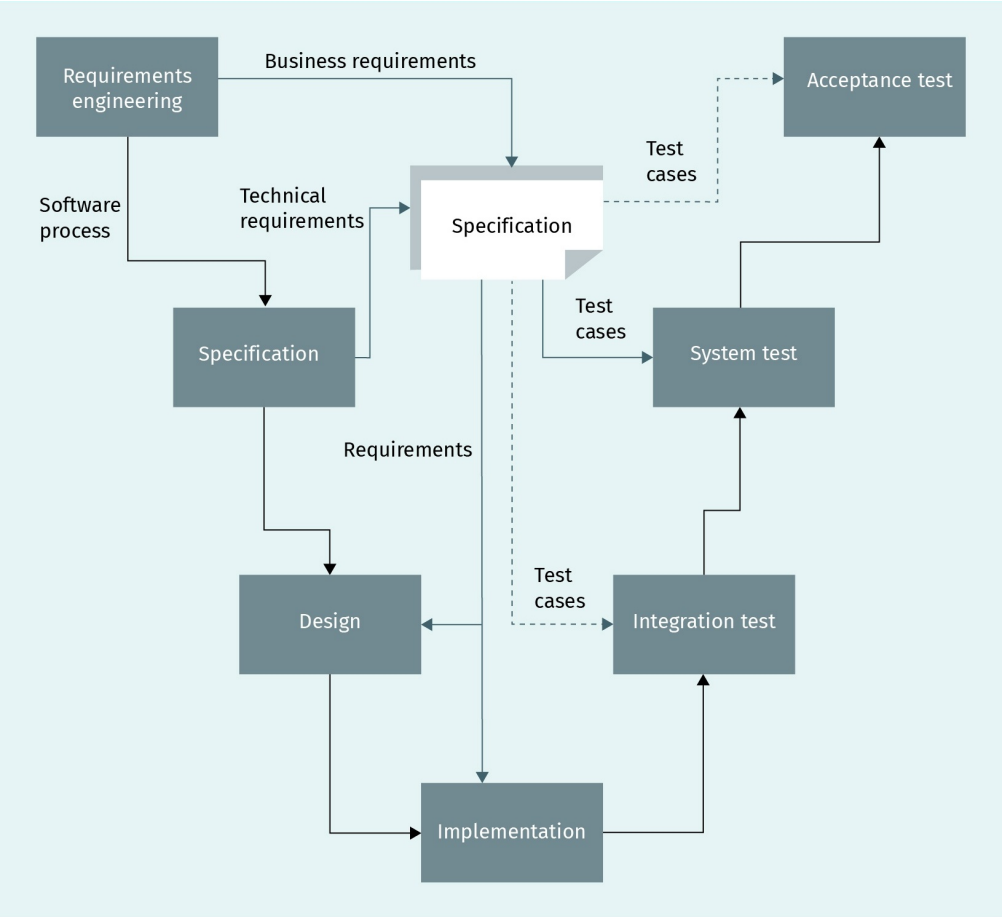
The technical specification is sometimes described as the result of RE, and no distinction is made between RE and specification. However, in this course book, we use the term "requirements engineering" to refer to the activities, methods, and techniques for determining, documenting, checking, and coordinating requirements that focus on the business perspective. In contrast, we use specification to denote the activities for the documentation of detailed technical requirements. In relation to the activities of requirements engineering, the specification is an extension of the documentation of requirements. There is no difference between RE and specification in terms of elicitation or validation techniques.

Usage of a Specification within Software Development

The figure below illustrates the use of specification documents within software development. Determined business requirements are expanded and refined in the specification to include technical requirements. The result is a business-technical specification, based on

which, the system design is created and the system is implemented. Test cases are also created based on the specification for the various test levels, in which the software system is tested for compliance with the requirements.

Figure 10: Usage of a Specification within Software Development



Source: Created on behalf of IU (2023).

Because the specification is both the basis for realizing the system (design and implementation) and the basis for the formulation of various test cases, mistunderstandings of or errors in the specification have far-reaching effects on the entire project. To specify the technical requirements as clearly and measurably as possible, there are many means of representation in software engineering (e.g., software models or notations). From a technical point of view, the specification of a system provides the technically detailed framework for design decisions. A specification does not decide how the system must be constructed internally, but only describes the externally visible system properties. From the point of view of the specification, the system is a black box of which the internal structure is unknown.

Specification Elements

A specification makes statements about what a system must be able to do. It thus describes the externally visible functional behavior of a system. Elements for specifying the functional behavior and properties of a system are as follows:

- data model. The data model contains the business objects that are processed in the system and their relationship to one another. Examples include the claim report, insurance application, and customer data.
- business functions. This is a functional description of the system's tasks or the specified component. Examples include an algorithm for calculating premiums and the procedure for terminating or concluding a contract.
- business rules. These are rules for a business object that must not be violated. Examples include that the start date of the contract must be before the end of the contract, and the total of the shopping cart must not be negative.

In addition to the externally visible functional behavior of the system, the technical system interfaces must be specified. The parts of a system that are used for communication with users or other systems (system environment) are referred to as **interfaces**. The following properties are specified for interfaces:

Interfaces
These are the parts of a system that are used for communication with users or other systems.

- the purpose of the interface (e.g., transfer of fund prices to fund management or validation of address data)
- the technical protocol or the technical rules according to which the system communicates with its environment (e.g., HTTP, FTP)
- in system interfaces, the data structure of the messages that are exchanged at the interface (e.g., XML, CSV)
- in user interfaces, the structure and behavior of the user interface

Quality requirements and constraints in the specification are refined on a technical level. Quality requirements of a system or a component are made verifiable, for example, through measurable metrics. An example of availability is as follows: "The debt collection system should be 99.9 percent available within one year. Scheduled maintenance work does not limit the availability." This means that around eight hours and 45 minutes of buffering are available for unscheduled maintenance.

The system must comply with technical and organizational constraints, such as guidelines, standards, laws, and style guides. An example is the guidelines for revision security, in accordance with national commercial or tax codes.

Specification of Graphical User Interfaces (GUI)

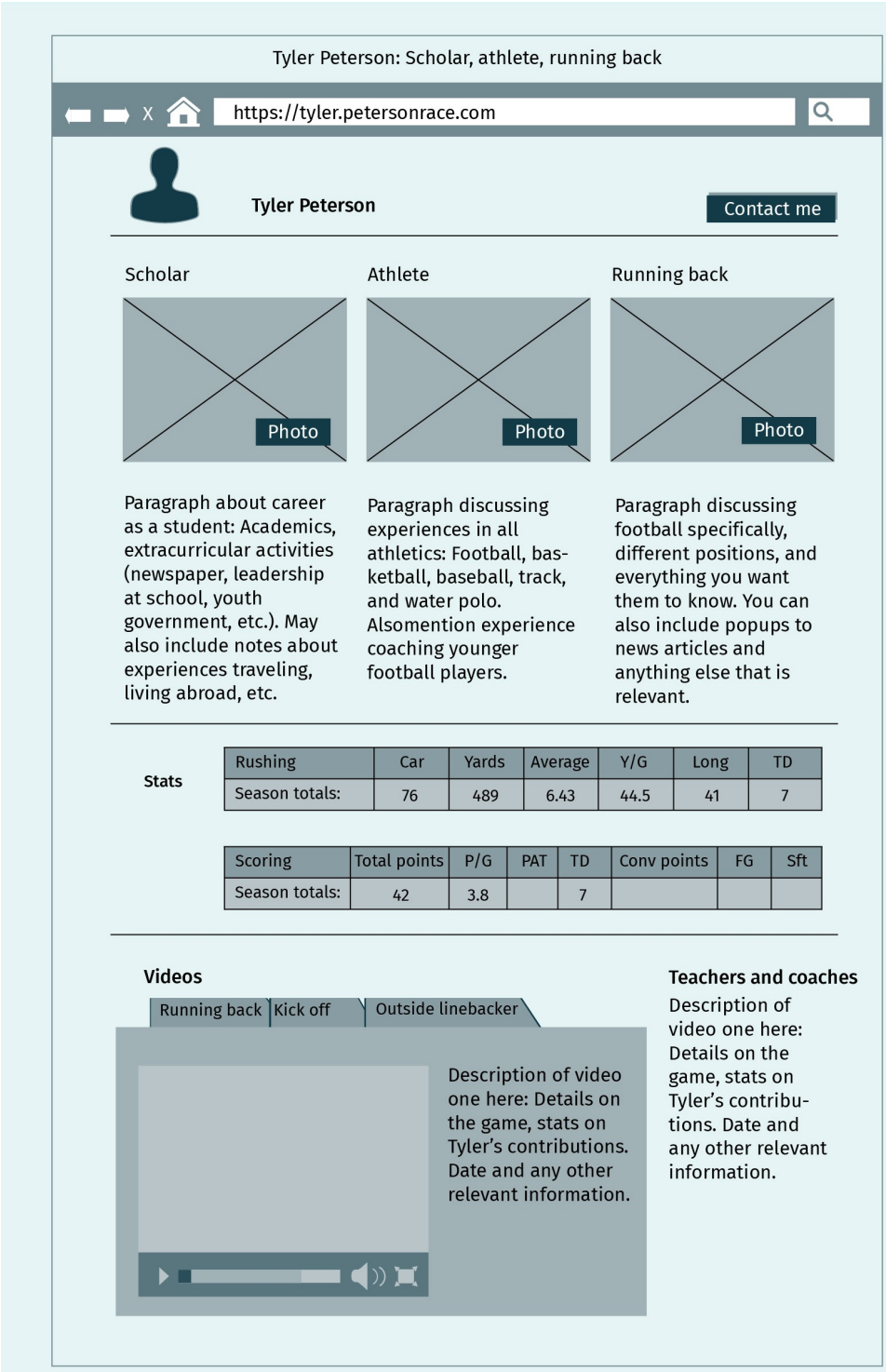
User interface specifications define specific requirements for the following aspects:

- contents and structure of individual dialogues, which are detailed specifications on type, size, position, color, and content of user interface (UI) elements, for example, input fields, texts, buttons, and images

- input validation, which is the specification of the rules to check input fields for business plausibility
- dialogue flow, which is the specification of how the user is guided through the UI, depending on the input and actions of the user

GUIs are specified both visually and textually. The visual specification defines the appearance of the UI elements used, their arrangement on a window frame, and the relationships to business objects and business functions. Such visual specifications can include sketches or screenshots of UI prototypes, as well as simple diagrams that show the dialogue flow. The textual specification details relationships that cannot be directly expressed visually or whose visualization would make the overview more difficult. These include the rules for activating and deactivating UI elements, validation rules, and dialog flow conditions. The figure below shows an example of a simple UI prototype.

Figure 11: UI Prototype Example



Source: Crawford (n.d.). CC BY 2.0

Specification of the System Structure and Behavior

The specification of the structure and the behavior of the system itself, as well as its technical interfaces, can be divided into four steps:

1. Identification and specification of functional components
2. Specification of business data models
3. Specification of business rules
4. Specification of technical interfaces for data exchange

Components

As a rule, a specification does not describe the software system as a whole. Depending on the specific project situation, only a very specific part of the system is relevant. Complex software systems are therefore divided into components, with each component being an independent software unit that can be combined to form a software system based on agreed interfaces with other components. As a result, specifying components means the following:

- identifying components and their relations to each other (e.g., shopping cart, article, and customer)
- describing functional responsibilities of components (e.g., extension of the term of the contract and calculation of the total amount)
- describing the processes and the logical structure of the messages exchanged at the interfaces of components

Business data models

With business data models, key entities of a business (or business objects) and their interrelationships are specified and defined uniformly for all parties involved. With the help of software models (e.g., unified modeling language [UML] class diagram) the structure, composition, and relationships between different business entities are specified. Based on business data models, the exchangeability of components can be guaranteed, especially in complex systems in which the same data are processed in different components. Moreover, data that are entered via a UI must correspond to both the data model, and the data stored in the database.

A business rule is a statement that defines or conditions a business aspect. Business rules can be used to describe assertions about business object properties and influence the behavior of business processes. In addition to structural requirements (e.g., “a contract must always be assigned to a policyholder”), business rules can also specify operational requirements (e.g., “damage over 500,000 dollars must always be assessed by two members of staff” or “contracts over 500,000 dollars are only concluded in the head office”). In practice, business rules are usually specified in the form of text.

Specification of Quality Requirements

In addition to the specification of functional requirements for software systems, quality requirements of the system must also be described in the specification. According to the International Organization for Standardization (ISO) standard 25010, the term “software quality” can be defined as “the degree to which a product or system can be used by specific users to meet their needs to achieve specific goals with effectiveness, efficiency, freedom from risk, and satisfaction in specific contexts of use” (The International Organization for Standardization, 2011, section 4.1). To specifically address the quality requirements, business requirements often state that the system must guarantee “good usability,” that “data security” should always be guaranteed, and that the system should “always” be accessible.

As part of the detailed specification of quality requirements, measurable quality criteria must be derived from the general requirements mentioned above. Quality requirements described in detail have a decisive influence on the architecture of the system in development. It is important to know whether the system should work with five or 500,000 users at the same time, or whether the system must be available on weekdays from 9 a.m. to 4 p.m. or on all calendar days 24 hours a day. Quality requirements usually have a stronger influence on the technical implementation than the expansion of the scope of business functions. In addition, particular attention must be paid to the testability of quality criteria. Again, specific test cases are created based on the specification. Therefore, the quality requirements must be specified in such a way that their fulfillment can be proven in tests.

SUMMARY

Before constructing parts of the system, which functions and properties the system should support after the development has been completed must be determined. The activities elicitation, documentation, validation, and managing of requirements are summarized under the term requirements engineering (RE). All relevant stakeholders of the system are involved in RE. By performing RE activities several times (e.g., iteratively), requirements for the system are determined and refined.

The specification describes the activities for the documentation of detailed technical requirements. The result is both a business and technical specification, based on which, the system is designed and implemented, and test cases are created for the various test levels. The software system is tested according to these levels for compliance with the requirements.

UNIT 5

ARCHITECTURE AND IMPLEMENTATION

STUDY GOALS

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

- define what architecture means in the context of enterprise software engineering and how architectures are applied.
- explain which typical activities are required to create a software architecture.
- describe how the architecture and the implementation of a software system are related.
- name the activities that are carried out during the implementation and explain how these can be supported with development tools.

5. ARCHITECTURE AND IMPLEMENTATION

Introduction

While requirements engineering and specification focused on descriptive core activities, software architecture and design are constructive activities. Here, based on the business requirements and the technical specification, specific determinations are made for the first time regarding how the system in development fulfills these requirements.

5.1 Architecture

In literature and practice, there is no generally recognized definition or concept formation for the terms “architecture” and “design.” Therefore, the term “architecture” is used differently depending on the organization, personal characteristics, and preferences. The term “IT architecture” can be used to denote the following:

- the process of designing software systems (and subsystems) of all kinds
- the name of a specific architecture typology, e.g., “client-server architecture”
- the professional field of the information technology (IT) architect
- the result for which an IT architect is responsible, i.e., the amount of artifacts they compose to create an architecture
- the term for the science of designing IT systems, i.e., the teaching of IT architecture

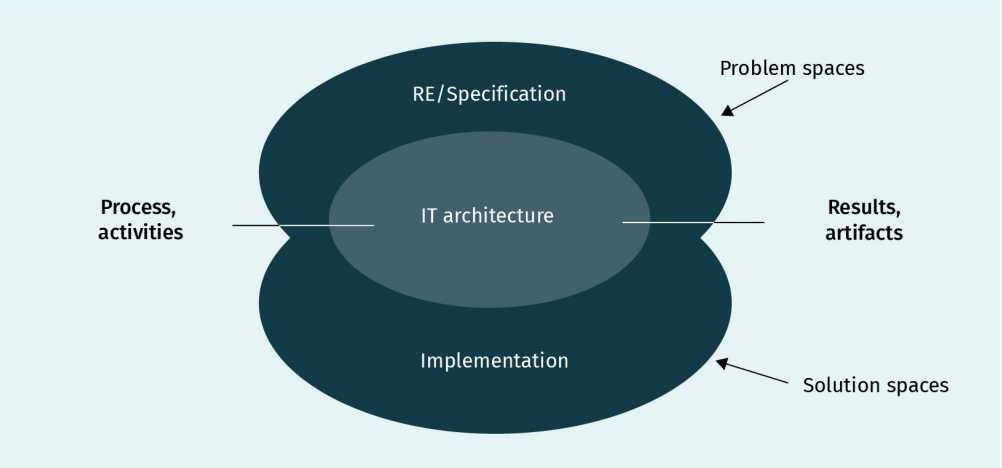
Both architecture and design are activities that decide how the future system will be structured and built up. Architecture is often used to describe the “preliminary design” of a software system, and design is used to describe the “detailed design.” Compared to the architecture of a house, this means that the architecture determines the basic structure, shape, type of roof, number of rooms, and the arrangement and shape of the windows. What the faucets look like, what color the tile joints are, and in which direction each door opens, is determined in the design. However, this term is not used uniformly.

Please note that in this unit, we use the term “architecture simplifying” to refer to all specifications and decisions about the system that are made prior to implementation. It is a collective term for everything that is also referred to as design.

From Problem to Solution

As depicted below, the architecture describes the transfer of the set of all requirements to a system (the problem space) into a concrete, executable software system (the solution space).

Figure 12: IT Architecture in Software Processes



Source: Created on behalf of IU (2023).

Starting with the requirements engineering (RE), system requirements are analyzed and specified. During the implementation, the set of conditions and specifications for a given system is translated into executable program code. The architecture is located between RE and specification on the one hand, and implementation on the other hand. The IT architect must analyze and understand the needs of the stakeholders, weigh them against each other, and develop an architectural description through several decisions and design activities. The architectural description represents the connection between the solution space and the problem space. It serves the development team as a template and framework for the following implementation activities.

Core Activities of Architecture Definition

The architecture definition activities, i.e., the process that leads to an architecture description, can be divided into three core activities.

Analyzing the requirements and interests of the stakeholders

First, the architect analyzes what is important for the respective stakeholder. In addition to the specified business requirements, they must also consider the stakeholders who are involved in the development and operation of the system, as well as the client. As a rule, the architect also gives an initial estimate of the cost, as they can estimate the complexity of the implementation of the specified requirements. Therefore, the architect helps and supports when balancing and making decisions between costs and functionality. Subsequently, the possibly restricted set of requirements must be documented and accepted by the stakeholders.

Designing an architecture that meets the requirements

Based on the set of requirements, the architect makes design decisions to meet the requirements. That is, they create a first version of the architectural description. The decisions must then be checked and evaluated against the requirements. If necessary, the IT architecture must be revised. If the result of the evaluation is positive, the IT architecture is refined and detailed until a sufficient solution has been found.

Describing and documenting the architecture

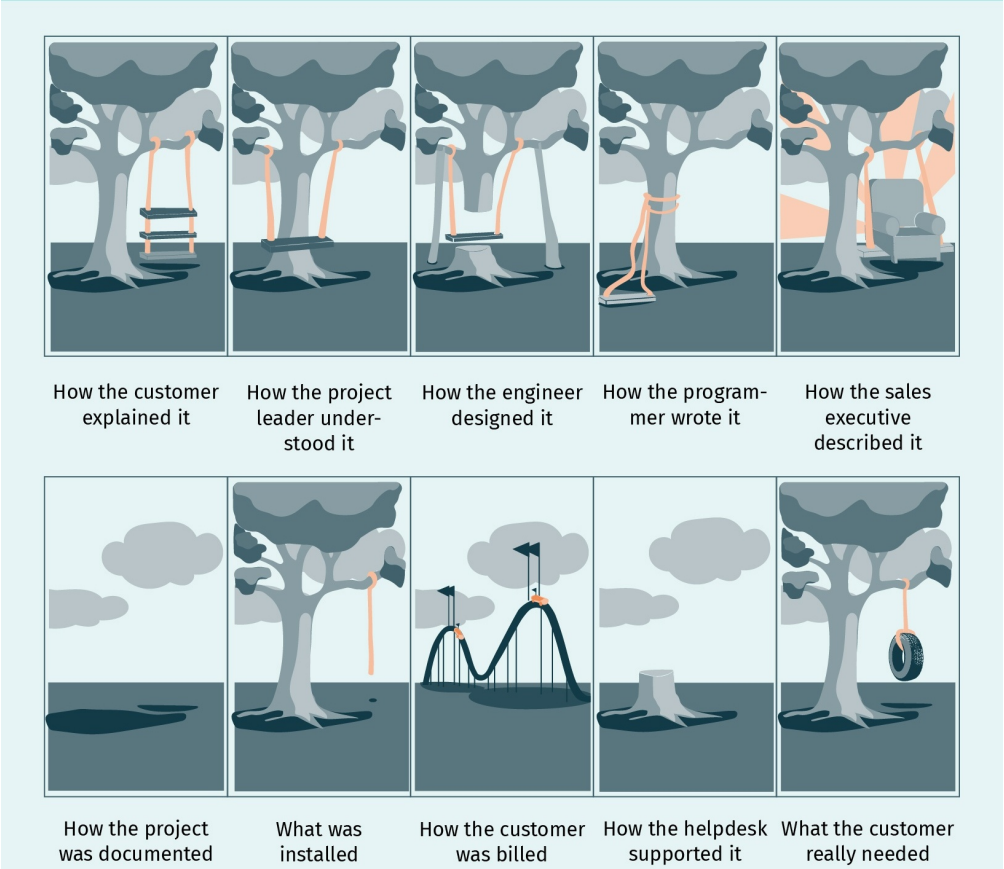
Finally, the decisions must be documented in the form of an architectural description. As with the activities of the specification, architectures are technically documented using software models. The more detailed the decisions about the architecture, the more technically and precisely the results must be documented. Of course, the architecture description activities can take place iteratively (in several cycles). Every additional requirement that is recognized during software development and classified as necessary must be assessed by the software architect and incorporated into the existing architectural description.

Architectural Description

The result of the design activities of an IT architect is the architectural description. However, the actual architecture of the system does not manifest until the system is implemented. Every system has an architecture, even if no software architect has created an architectural description. Architecture is therefore inherent in the system. You can also build a house without consulting an architect. The finished house then has an architecture, although it happened arbitrarily. Of course, such an architecture may not be beautiful or practical. This concept can be transferred to software systems. In principle, a system can also be implemented without an architect. However, as with building a house, there is a risk that important elements will be forgotten or placed inappropriately, or that the basic structure of the system cannot withstand the actual loads due to, e.g., high numbers of users.

IT architectures can be documented in different ways. Simple sketches and presentation slide graphics are often created for initial ideas, as shown in the figure below. Although these can be created quickly, they have no clear and agreed-upon semantics and often allow a dangerously large scope for interpretation. This type of representation is generally not understandable for people who are not involved in the creation process.

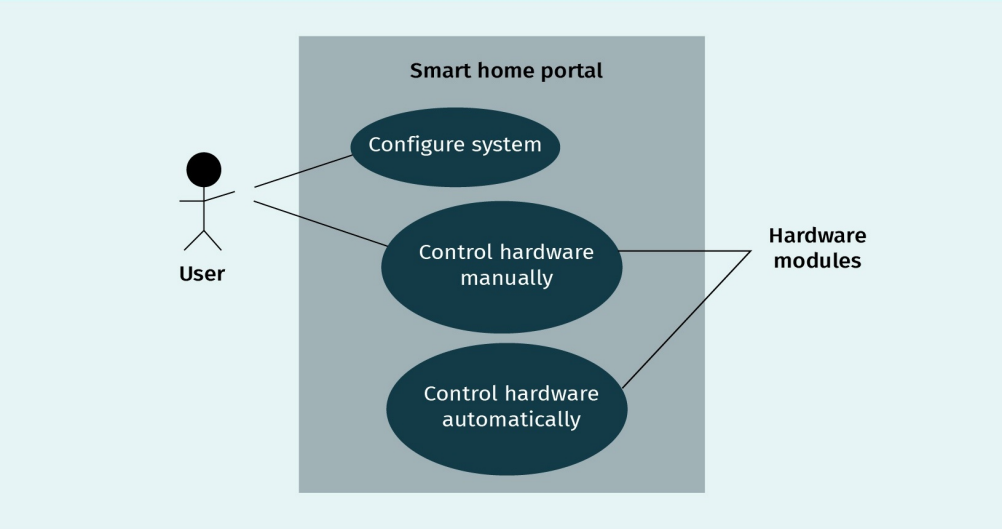
Figure 13: Tree Swing Cartoon



Source: Created on behalf of IU (2023).

Therefore, architectural descriptions should always be documented with well-defined software models (e.g., unified modeling language [UML]). Their notation elements have a clear meaning (semantics) and are used worldwide. In the meantime, almost every IT graduate gets to know the UML models in their studies or training, so the use can be recommended without restriction. An example of documentation with UML is shown in the figure below.

Figure 14: UML Use Case Diagram



Source: Created on behalf of IU (2023).

A major challenge in dealing with architectural descriptions is the divergence of the documented design decisions in the form of the architectural description and the actual system architecture. Software developers are neither physically nor logically bound by the architect's specifications. There are essentially no restrictions in programming, so the expressiveness of a programming language can be used by the developer at any time, even if it contradicts the original architecture specifications.

Application Scenarios for Architecture Documentation

There are two different possible uses cases of architecture documentation: *a priori* and *ex post* documentation. **A priori documentation** is the creation of an architecture description before the implementation of the system based on design activities and decisions. In this case, the architecture documentation is used to express and document agreements about the design of the system. It defines the framework within which the development teams can make their own decisions, and thus the target state of the software architecture. The *a priori* documentation also forms the starting point for the validation of the suitability of an architecture to support required quality requirements and constraints.

A priori documentation
This is the creation of the documentation prior to the implementation of the system.

Ex post documentation is the creation of the architecture description after the implementation of the system, based on the implemented program code. It is used to represent and document the architecture that has been implemented by the development team. An architecture description generated in this way represents the actual state of the software architecture. It can either be used to check the conformance of the implemented architecture against the architecture, or as a starting point for the architecture evaluation with regard to compliance with standards, guidelines, or laws.

Ex post documentation
This is the creation of the documentation after the implementation of the system.

Architectural Levels

In principle, architectures can be created on different levels in software engineering. There are various IT architectures, including the overview of business concepts and dependencies using a business architecture, the representation of the structure of individual software systems using a software architecture, and the representation of the entire enterprise IT architecture. In principle, a specific architecture can be created for each relevant sub-area or each special view of a software system, e.g., a security architecture or a communication architecture. The table below compares the various levels of IT architectures (business architecture, software architecture, and IT enterprise architecture).

Table 3: IT Architecture Layers

	Business architecture	Software architecture	IT enterprise architecture
Subject	• Documentation of business elements, actors, and processes, as well as their dependencies and relationships to another	• Specification of the structure of a software system, within which software developers implement the program code • Documentation of the actual structure of a software system	• Documentation of all IT applications used in the company and their relation to (value-adding) business processes • Documentation of development planning • Monitoring and changing processes
Elements	• Business entities (data and business objects) • Applications (services) • Actors • Business processes	• Business entities (data, business objects, interfaces, systems, and components) • Applications (web services and subsystems) • Technical processes within systems • Frameworks	• Applications • Business entities • Business processes • IT infrastructure (data center and networks) • Processes • Committees of architecture management
Areas of application	• Project organization • Requirements engineering • Training • Documentation	• Architecture • Design • Implementation and documentation of software systems, including re-engineering and evolution	• Alignment of IT with corporate goals • Management of the IT application portfolio • Specification of framework conditions for the creation or procurement of software systems
Notation	• Box-and-lines diagrams • Business process-related descriptions (BPMN) • UML	• UML (class and component, activity, and sequence diagrams) • Box-and-lines diagrams	• Box-and-lines diagrams • Process maps • Business processes • Application manuals

Source: Created on behalf of IU (2023).

5.2 Implementation

The actual creation of the software system takes place through the implementation, i.e., the writing or generation of program code. Based on the documented business requirements, the technical specification, and the architecture description, several program code artifacts are created, which, together, make up the executable software system.

Dependencies on Architecture

Transferred to the construction of a house, the implementation corresponds to the trade workers, such as bricklayers, carpenters, or plumbers, who carry out the tasks assigned to them according to the specifications of the architect. As with house construction, there are more executing (implementing) people than architects in enterprise software development. The spectrum of tasks for software architects includes overall responsibility for the entire system, responsibility for individual components, and responsibility for individual interfaces, depending on the specific project. The software architect can also be actively involved in the implementation, depending on the project organization.

In any case, the architect is the first point of contact for the software developer for questions related to the business or technology, and if decisions will affect several parts of the system or important interfaces. If software architects are actively involved in the implementation, they can make sure that the developers adhere to the specifications of the architectural description. This reduces the risk that the architecture will develop away from the planned architecture during implementation (architectural erosion).

Creating Program Code for a System

Essentially, a software system manifests itself in two ways:

1. The source code (also known as program code, code, or code base) and the number of artifacts (e.g., configuration files, media) created by the development team
2. An executable that can be deployed and started on a computer, which is usually a binary file or package that has been compiled from the source code. In the context of cloud and distributed computing, the deployment can also take place on a virtual machine, a computation cluster, or a cloud service.

The activity of writing program code is called programming (also known as implementing and coding). When creating the code based on the architectural description, software developers can ensure, in various ways, that the system supports the required functions. This happens either through writing program code, reusing existing program code, or generating code automatically. In practice, all three types of source code creation are combined.

Writing code

When writing program code, the programmer creates structured text for the targeted solution of specific problems. For example, they create a data structure, implement an algorithm, or program rules for processing data. When creating, they are firmly bound to the means of expression and the words of the programming language in use.

Reuse existing code

Depending on how widespread a programming language is, and whether the software developer must solve a very specific technical problem or a general and recurring problem, the developer can fall back on already programmed solutions to the problem. Solutions to standard problems, such as creating a data structure that can store, sort, and search through various items, are supported by almost every programming language today. Existing solutions to general problems are made available to developers for reuse in libraries or frameworks. These libraries are either part of the programming language or are offered by third parties. In addition to commercial providers, there are countless libraries, especially for the Java or JavaScript programming language, which are made freely available under an open-source license.

The creation of complex information systems that can be accessed via a browser or app and connected to many other systems is inconceivable without reusing existing code in the form of frameworks or libraries. So, in addition to the ability to write program code, a fundamental skill of developers is knowing which framework and which library provide which functions, and how these functions can be integrated with each other and with program code that you have written yourself.

Like the make-or-buy decision in the planning phase, developers and architects must also decide whether to produce code themselves or reuse it to solve specific implementation questions. Frequently used and already tested program code from a freely available library is usually preferable to developing it in-house.

Generating program code automatically

Another possibility to generate program code is the automatic generation of code from elements of the architectural description. Depending on how detailed an architecture description is, program code for database connections can be generated directly from the data models, for example. This relieves the developer of the manual creation of technically trivial program code. Another typical application is the generation of program code for user interfaces by graphic editors. Overall, the generation of program code makes sense if the effort for configuring and maintaining the code generator is less than for the manual generation of the code fragments.

Development Environment

Software development in an enterprise context always takes place in a team. Several developers are therefore always working on the code base of a system at the same time. In order to give the development team the best possible support and automate trivial tasks

Integrated Development Environment (IDE)
This is a set of software tools or applications used to provide comprehensive facilities for software development.

as much as possible, an appropriate development environment (**integrated development environment** [IDE]) is required for software development. The key functions of an IDE are described below.

Code and libraries

An IDE provides support for writing program code, for example, through syntax highlighting, recognition and display of compilation errors, step-by-step execution for debugging, refactoring, code completion, navigation through the program code, and simple compilation and execution of the program for test purposes. It also enables the integration of libraries and framework through simple import and navigation functions, and the generation of program code for trivial tasks, for example, with a graphical UI editor and special dialogs for generating standard code fragments.

Version control

Several developers must be able to work on the same software system at the same time. Therefore, it must be possible for every developer always to have an up-to-date version of all code fragments. No one should be able to accidentally overwrite the work of colleagues, and a current version can be restored at any time in the event of unintentional deletion. GitHub is the current system of choice for version management.

Continuous build and integration

In large software projects, several dozen or hundreds of developers work on the parallel implementation of various components using many external libraries and frameworks. Nevertheless, each individual developer must be able to generate a current version of the program under development to test the changes that have been made. With toolchains (pipelines) containing build tools like Gradle or Maven, the compilation and integration of source code into complex software packages can be automated. In addition, automatic tests can be started with the use of build systems, which test the automatically compiled system version for errors before delivery.

Typical IDEs that are suitable for working on large enterprise software projects are, for example, IntelliJ IDEA, Eclipse, or Microsoft Visual Studio. Other IDEs are specially designed for a very specific purpose, such as Android Studio and XCode for app development.

Programming Languages

The software system is implemented in a specific programming language. The figure below shows the ten most frequently used programming languages, based on data regularly published in the The Importance of Being Earnest (TIOBE) index (TIOBE, n.d.). C is one of the oldest languages in the top ten. It is often used for hardware-related programming, for example, in embedded systems. Python is used in many contexts, like the development of back-end software, data science, machine learning, and embedded systems. Java, C#, and C++ are the object-oriented programming languages. For the past two decades, these languages have been continuously used in the development of enterprise soft-

ware, especially for information systems. However, Java is also used for programming apps since Android smartphones became available. JavaScript is used in the field of front-end web applications. Together with back-end languages like Java, Python, or C#, JavaScript enables interactive user dialogues. In addition to such general-purpose programming languages, with which, theoretically, any type of software can be created, there are also programming languages for very special areas of application. Saving and reading data in a database is often implemented using structured query language (SQL).

Table 4: TIOBE Index (June, 2021)

Ranking	Programming language	Rating (%)
1	C	12.54
2	Python	11.84
3	Java	11.54
4	C++	7.36
5	C#	4.33
6	Visual Basic	4.01
7	JavaScript	2.33
8	PHP	2.21
9	Assembler	2.05
10	SQL	1.88

Source: Benner-Wickner (2021), based on The Importance of Being Earnest (n.d.).

In addition to the languages that are programmed and executed, there are also markup languages (hypertext markup langauge [HTML], extensible markup language [XML], cascading style sheets [CSS]). These languages are typically used to describe the structure and content of application interfaces (websites). Markup languages can be used to specify a structure in which messages can be exchanged between systems, but not how systems process this information.

Different programming languages can therefore be used in a software system, for example, HTML and JavaScript for interactive UIs, Python for business logic, SQL for reading and storing data in the database, and XML for exchanging messages with other systems. To combine the programming languages and use them with one another, the developer usually has libraries and frameworks at their disposal that make integration easier for them.

As a rule, the software architect decides which programming language will be used to implement a specific software system. These decisions are constrained by the chosen technology. For example, the enterprise resource planning (ERP) system, systems applications and products in data processing (SAP) has to be customized using the advanced

business application (ABAP) language. In the case of enterprise information systems that must be integrated into a given application landscape, various factors must be taken into account when determining the language:

- Which runtime environment will be available later in operation?
- Which languages do the available developers know?
- How does the language fit into the application landscape?
- Which technologies or languages are used to implement the applications within the system context?
- Is it ensured that the knowledge about a technology is available over the entire planned system life cycle?
- Are there already mature frameworks and libraries for typical problems?
- Is there sufficient experience with the language in the use of productive systems comparable with the system development?
- Are tutorials, trainings, and experts available for this language?
- Has language support been secured for future system platforms?

The more mature a programming language, the more libraries, know-how, and support are available. Therefore, it is usually very rare that business-critical information systems are implemented with newer programming languages. Compared to IT departments in large companies, smaller companies and start-ups are often more willing to experiment with programming languages. In specific safety-critical branches, such as aerospace and automotive, the use of International Organization of Standardization (ISO)-certified programming languages like C and C++ is mandatory.

SUMMARY

Architecture is located between specification and implementation. The IT architect must analyze and understand the needs of the stakeholders, weigh them against each other, and develop an architectural description through several decisions and design activities. The architectural description serves the development team as a template and framework for the implementation.

The source code of a system is only programmed during implementation. Supported by a development environment, the development team creates the source code of the software system by manually writing, reusing existing code, and automatically generating code. The choice of programming language heavily depends on the organizational and technical constraints under which the system is to be developed.

UNIT 6

TESTING, OPERATION, AND EVOLUTION

STUDY GOALS

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

- define quality assurance and management in the context of software engineering.
- understand what test levels are and outline why they are needed.
- explain which activities are required in the operation of enterprise software systems.
- discuss what dependencies there are between the core activities.
- evaluate the challenges in the evolution of software systems.

6. TESTING, OPERATION, AND EVOLUTION

Introduction

Although a software system can only be tested holistically for compliance with the necessary requirements after completion, the risk of waiting until the end of implementation to perform quality assurance activities is too high. Therefore, quality assurance measures in software engineering are carried out alongside all other activities. After installation, acceptance, and provisioning of the software system, the application operation ensures that the system is available to users and that it is protected against failure and threat scenarios. The application operation itself has no explicit constructive or quality assurance function in the development of a software system. However, when the software is handed over, it must be guaranteed that it can be executed in accordance with the company's specifications and guidelines. The software must also be able to be taken over into operation by the department responsible for information technology (IT) operations after development and testing. After the first release of a software system, adjustments and changes must be made to the system over the entire period of use. If the system is to be expanded by technical functions, or if technical functions are changed, these are activities of software evolution. This unit will introduce the core concepts within software testing, including software quality and quality management.

6.1 Testing

Software Quality

Software quality is the "degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value" (International Organization of Standardization [ISO], 2011, section 4.1). Accordingly, one would have to check software to determine whether what has been delivered meets the previously specified requirements.

The importance of a specification becomes clear: According to ISO, decisions about the quality of software can only be made on the basis of the specification ("stated needs") (ISO, 2011). In practice, however, it has been shown that the perceived quality of software is primarily determined by whether the customer determines that the software meets their actual requirements ("implied needs").

Since many requirements are often only recognized during software development, with a view to customer satisfaction, it must be continuously ensured that the set of specified requirements also includes the set of actual requirements. In addition to determining the quality of a software system as a whole, the quality can also be determined for the artifacts (software models, requirements, architectures, documents, software components) generated in the course of software development.

Quality Management

Quality management
This is the summary of all organized measures that serve to improve quality.

The term **quality management** (QM) summarizes all organized measures that serve to improve the quality of products, processes, or services of any kind. Since it is too risky in software engineering to wait to determine the software quality until implementation is complete, the quality of software fragments that have already been created is determined during the development process. However, the generated program code is not the only aspect that must be considered in software quality management. Because the implementation is directly dependent on the specifications and decisions made in the requirements engineering, specification and architecture activities, errors in the specification, and the architectural definition spread to the implementation. Therefore, these artifacts must also be included in quality management in software engineering.

Typical activities in quality management are as follows:

- quality planning. This is the creation and documentation of the quality requirements together with the client.
- quality control. This is monitoring, management, and control of activities for quality testing in the software development process.
- quality assurance. This comes in the form of activities ensuring that defined quality requirements for products, processes, and services are met.
- quality improvement. This is the evaluation of product and process data to improve the quality level.

Constructive versus Analytical Quality Management

In principle, quality management can be divided into constructive and analytical quality management. In constructive quality management, all quality properties for products or processes are defined *a priori* (i.e., before creation) in order to avoid errors during software development and to guarantee or increase the quality of the artifacts created. With forward-looking, constructive quality management, the effort in analytical quality management can be reduced. The measures for constructive quality management are

- technical (e.g., use of modeling languages, tools, development environments),
- organizational (e.g., guidelines, standards, templates, checklists), and
- socio-psychological (e.g., training, working atmosphere, team building activities).

In analytical quality management, measures are carried out *ex post* (i.e., after creation) to test and evaluate the current quality level of the test objects in order to systematically track down errors and determine their extent. A further distinction can be made between static and dynamic analysis procedures. These terms are defined as follows:

- static analysis. As part of a review or audit, the test item is analyzed, appraised, and examined. The information obtained is collated, if necessary, condensed into metrics or key figures, and evaluated.
- dynamic analysis. The system under test (SUT) is executed with specific input values, and the result of the execution is evaluated.

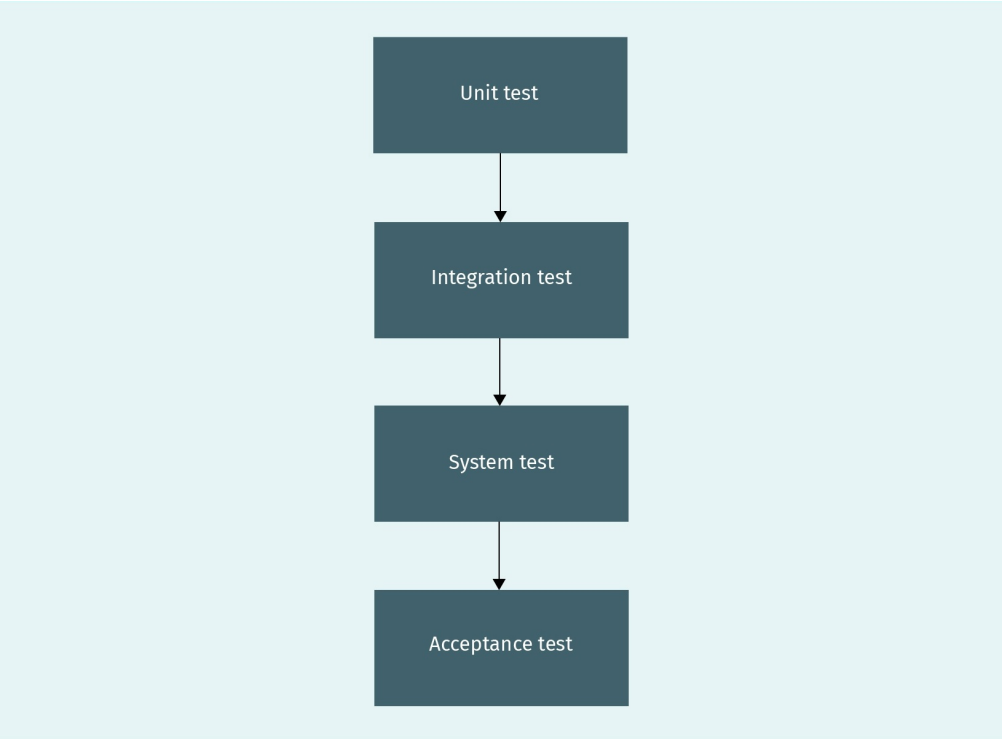
Test Item

In principle, all artifacts created in software engineering can be tested for compliance with quality requirements. This means that business requirements, technical specifications, architecture descriptions, and test cases can be tested in addition to the generated program code.

Test Levels

When testing the generated software system, a distinction is made between test levels (also known as test stages). It is too risky to wait until implementation is complete to perform software quality assurance. However, before the system can be tested in its operational environment, test activities are started in enterprise software development as soon as the first software artifacts have been generated. The figure below shows a schematic overview of the various test levels in software engineering.

Figure 15: Test Levels for Software Quality



Source: Created on behalf of IU (2023).

Unit test

To enable the implementation of a software system by several software developers, the system is broken down into its logical components (units, modules, classes, and building blocks). Each component is implemented separately and incorporated into the system after completion (integrated). During, or after, the completion of a software component, it

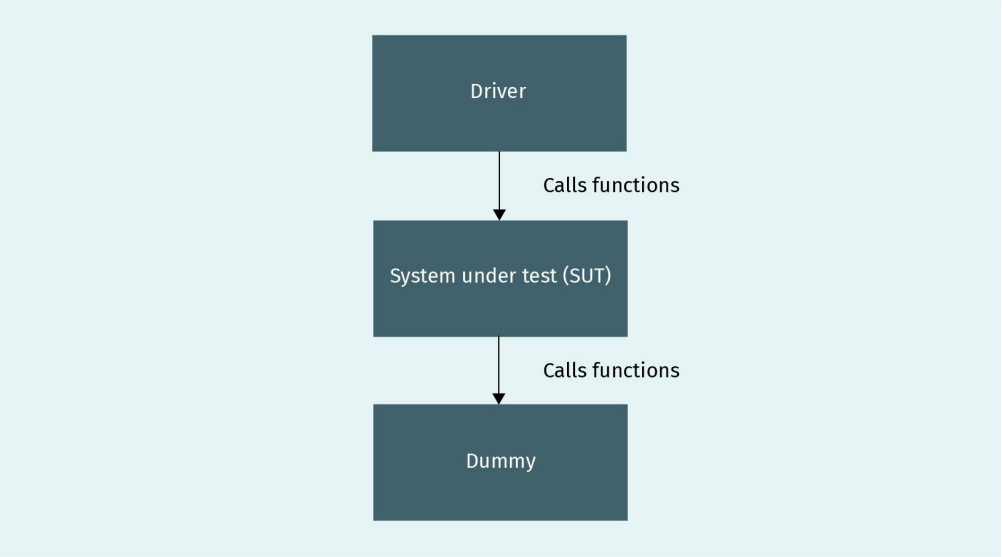
can be checked for compliance with specified requirements. The isolated testing of individual software components is called a unit test. Some examples of unit tests are as follows:

- premium calculation requires calculation of insurance premiums depending on the selected services
- shopping cart requires calculation of the total of all items
- address component requires the correct address for correspondence to be generated from the customer data

(Component) integration test

As soon as at least two software components have been completed, they can be merged into one system, i.e., integrated. During, or after, the integration, the interaction of groups of components is tested in integration tests. The integration tests check whether the components work together as described in the specification. In the case of complex software systems with many components, not all components are ready at the same time. In these cases, missing components are simulated by drivers and dummies so the test can be started. The figure below illustrates the interaction of components, drivers, and stubs.

Figure 16: Drivers and Stubs



Source: Created on behalf of IU (2023).

Software fragments that simulate the calling of other components are called drivers. Stubs, on the other hand, simulate components that are called by other components. Technical interfaces for external systems are simulated in the integration test using stubs and drivers. The interaction of the interfaces must be tested beforehand because only the finished software system can be connected to external systems. Some examples of integration tests are as follows:

- integration of an address component with an external service for address validation

- integration of a portable document format (PDF) printer and the shopping cart to generate the invoice
- integration of the existing system with a debt collection system to automatically dispatch invoices

System test

After completion of the development work and the integration of all software components into a finished system, the system as a whole is tested with the system test. The aim of the system test is to check whether the system meets the specified requirements. In addition to the business requirements, the specified quality properties are also tested. In the course of the system test, targeted stress and load tests are used, among other things, to check the system behavior under particularly demanding situations (e.g., many users and large amounts of data, over short or long periods of time). The system test is usually carried out without the involvement of the client and is the last test completed by the organization that created the system before the software is handed over.

A major challenge in the system test is reproducing the customer's productive environment as true to the original as possible. This includes the hardware and software environment in which the system is operated, the software systems with technical interfaces used by the system, realistic utilization, realistic user behavior, and real data records. The provision of true-to-original datasets, which, at the same time, comply with data protection regulations, is a major and often unsolvable problem. Some examples of system tests are as follows:

- testing a new shop system after the old system has been completely replaced
- replacing a damaged system from a composite insurance

Acceptance test

The last test level is the acceptance test, in which the finished system is handed over to the client, installed, and tested under the actual operating conditions. The acceptance test checks whether the system has the contractually agreed performance characteristics from the client's point of view. The client often carries out the acceptance test, or is involved in the execution. In terms of the type of implementation, the acceptance test is a special type of system test; a clear distinction is usually difficult. The most important difference compared to the system test is that it is carried out by a different organization than the one that developed the system. A successful acceptance test is often the contractually agreed-upon requirement for invoicing.

Agile Testing

Early knowledge, which is valuable for Agile approaches in general, is gained through short feedback cycles. For this purpose, in contrast to traditional plan-driven approaches, only a small subset of the requirements are analyzed, specified, implemented, and tested by the development team from the start of the project. Scrum teams can, for example, derive test cases directly from the sprint backlog in order to test the planned partial result (called "increment" in Scrum). The abstraction level of the test cases is based on the

abstraction level of the requirement. For example, rough requirements are more likely to be checked with system tests, while detailed requirements can even be covered by individual unit tests. Because Agile teams are typically composed in an interdisciplinary manner, they can cover this range of tests well.

The individual tests can be carried out quickly because only the software part developed in the current sprint needs to be tested (plus regression tests, of course). This way, the team can quickly gain new insights into the actual requirements. The team can also determine early on in a sprint retrospective, for example, that the process has to be fundamentally adjusted and thus adapt to volatile constraints. Delivering early insights is therefore the central goal of agile testing. This goal results in the following characteristic features of agile testing:

- Test as early as possible and as often as necessary.
- As many steps as possible are to be automated (except for exploratory tests by experts)
- Concentrate on testing the basic functionality and select other test types according to actual needs (first make it run, then make it fast).
- Working in an interdisciplinary manner means, depending on the type of test, also involving the developers.

Setting up a tool infrastructure alone does not enable an IT organization to test in an Agile manner. For Agile testing in all test levels, the following organizational preparations must also be made:

- relocation of the test effort (less manual, more automated unit tests)
- shifting the test tasks to the developers
- the tester having disciplined work organization
- well thought-out test data management
- integration of development and operation
- integration of the user in the tests

6.2 Operation

Since enterprise information systems are business-critical operating and production resources for many branches and companies, changes to running systems must be planned and implemented with the utmost care. In particular, since modern information systems have technical and process-related dependencies on other systems in a company's application landscape, failures or defects in IT systems often affect the business capability of companies.

Release Planning

To carry out updates and changes to the company's IT under these circumstances, release plans are agreed to between the business departments, IT application development, and IT operations. A release plan defines specific dates on which change requests and adjustments to software systems are implemented and put into operation. With the help of

release planning, business departments, application development, and operations can internally plan and carry out the tasks required to import a new software version. This includes, in particular, quality assurance activities.

For large systems, between one and four release dates per year are common, and the specific dates are often based on industry-standard cycles or legal requirements. For example, online mail-order business is booming in the run-up to the winter holidays, motor vehicle insurance is often canceled in late autumn, online auctions usually end on weekends, and systems for clerks in authorities are only used productively on working days.

However, a clear release plan with, for example, four release dates per year is challenging. First, to accommodate all desired adjustments and change requests in a few releases, changes must be carried out at many points in the application at the same time. However, large adjustments always involve a high risk of errors. Therefore, comprehensive quality assurance measures must be carried out, which, in turn, leads to less time for the installation of the adaptations. Second, business departments are dissatisfied with the IT support, and important change requests are usually only implemented after weeks or months. Particularly in companies in which information systems are directly involved in the value chain, a slow IT response time can result in competitive disadvantages. Third, taking a long time to correct newly discovered security gaps or bugs of any kind in the application leads to open weak points in the IT infrastructure and to possible attack scenarios.

Continuous Integration, Delivery, and DevOps

To avoid the disadvantages of the few large releases without endangering operational reliability, daily and weekly releases of software systems can be created and automatically tested through automation and tool support. This is known as continuous integration, since all changes are integrated into an application in short cycles, regardless of specific dates.

If all tests are completed positively, the adapted application can be deployed directly by setting up an automatic delivery. Similar to continuous integration, “continuous delivery” puts the new version of an application into operation, regardless of specific dates, which is why the term continuous delivery is also used. The prerequisites for continuous integration and continuous delivery, however, are intensive cooperation between application development and operation, the organizational and technical ability to carry out regular automatic tests, and the organizational and technical ability to automate the commissioning activities. Such cooperative approaches are also called development operations (DevOps).

IT Service Management

In addition to the involvement of the departments responsible for the operation of IT systems in the planning and implementation of software releases, another task in operation is the provision of IT services. IT services include the provision and orchestration of data

networks, server systems (hardware), operating systems, storage systems, and backup solutions. In other words, they include services that are required for the development and operation of software systems.

To adapt the IT infrastructure to the specific needs of the business and IT users while keeping the effort for the operation and support of the IT infrastructure as low as possible, active IT service management must be carried out during operation. A global standard for IT service management is the Information Technology Infrastructure Library (ITIL) (Agutter, 2019). The ITIL categorizes and describes typical activities and best practices for managing IT services.

An important goal of IT service management is the development of a standardized service portfolio that is specifically tailored to typical IT service requirements, but, at the same time, enables standardized and efficient operation of the IT infrastructure. Extensive standardization of the IT infrastructure is only possible if both technical requirements and architecture decisions in software development adhere to the framework conditions specified by the standardized IT service portfolio.

6.3 Evolution

Further development activities that make the system evolve are essentially the same tasks and activities needed for the initial creation of a software system. However, the evolution activities relate to a productively used system to which changes must be made.

Challenges of Software Evolution

This reference to existing program code brings the following challenges for the evolution activities:

- The internal structure (architecture) of existing systems degenerates with every change in the program code. Over the course of several years, software systems lose their originally planned structure, and the effects of even small adjustments cannot often be clearly assessed.
- The architecture and the technologies used in the system in development do not support the new requirements.
- The documentation of the system to be adapted is incomplete and not up-to-date. The specific parts of the program code that need to be changed can only be found through reengineering, which involves great effort.
- Lack of knowledge about the architecture and programming language of the system means that the development team is not able to reliably make changes.
- Existing functions that should remain unchanged must not be impaired during evolution. If a function is accidentally changed after an intended change, this must be recognized and rectified before the new version is delivered. Errors that have already been corrected may not be reintroduced in the new version.

Architecture and Documentation Governance

Activities for the adaptation and evolution of software systems are often carried out under the stipulation “quickly integrate and deploy.” Under the time pressure of the release plan, the elimination of the last errors before delivery is typically prioritized over updating documentation or maintaining of the system architecture (the internal structure). This accelerates the process of degenerating the system and ensures that future training, adaptation, and testing efforts are higher than necessary.

If the ability to evolve over a long period of time continuously be guaranteed, the software architecture is monitored and maintained continuously and in a structured manner during the many smaller adjustments. In addition, the system documentation must also be continuously updated. Identifying the points at which adjustments are made can easily take up almost half of the time needed for the evolution activities.

SUMMARY

In quality assurance activities, the artifacts generated during software development (program code, documents, models, and test cases) are checked to determine whether they meet specified requirements. Measures for constructive, as well as analytical, quality assurance are used. The quality assurance of program code takes place at different test levels, depending on the current status of the project. In Agile testing, these levels recur quickly in a massively automated testing environment.

When operating software systems, it must be ensured that the system is available to users and is protected against failure and threat scenarios. Changes to running systems must be planned with the utmost care and carried out with the help of release plans. In addition to release planning, IT service management activities are also part of operations.

After the first release of a software system, adjustments and changes must be made to the system over the entire period of use. During the evolution activities, the system is expanded to include business functions, or existing business functions are changed.

UNIT 7

ROLES IN SOFTWARE ENGINEERING

STUDY GOALS

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

- define the idea of the role-based approach.
- identify the typical roles in enterprise software engineering.
- explain which tasks and responsibilities the individual roles have.
- explore the dependencies that the roles have on one another.

7. ROLES IN SOFTWARE ENGINEERING

Introduction

Software engineering comprises many phases and activities that require distinct expertise in very different fields. In practice, this amount of deep knowledge cannot be known by a single person and is typically shared among many individuals. Hence, a large number of people are usually involved in the creation and operation of complex IT systems for use in large companies. The team size can range between two and several hundred people; there is no typical team size.

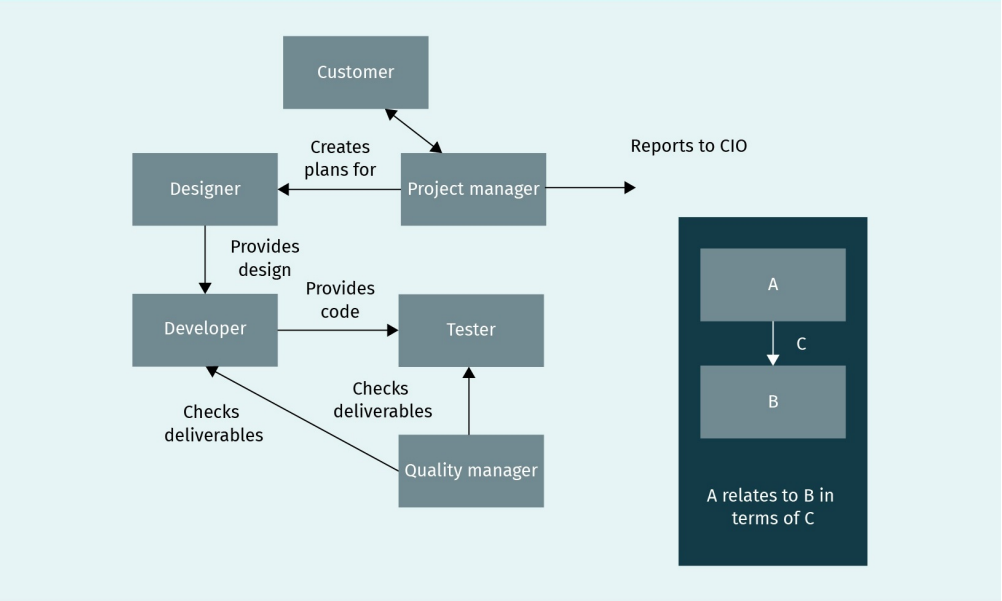
In this context, it can be a great challenge to divide up and control the activities of each individual team member in a meaningful way, especially with very large teams. For this reason, among others, individual people are assigned to roles in software engineering (Kruchten, 2003). Examples of such roles are requirements engineer, developer, architect, and test manager. One advantage of this type of division of labor is the possibility of specialization and expert training.

7.1 Idea of the Role-Based Approach

Role
This is a defined function to be performed by a project team member, such as testing, filing, inspecting, and coding.

A **role** is characterized by its name, specific tasks, and specific objectives. A person involved in the project has at least one specific role. Depending on the size of the project, roles are assigned multiple times. As a rule, there are always several people with the developer role. However, one person can also have several roles at the same time. A "developer" can also take on the role of a "tester," and an "architect" can also be a "developer." The figure below illustrates some roles and their relationships in an arbitrary example.

Figure 17: Example of Roles and Responsibilities



Source: Created on behalf of IU (2023).

With the explicit division into roles and the determination of objectives for each role, conflicts of objectives are automatically induced between roles in the project—and thus also between the people involved in the project. While the project manager must ensure that the budget and schedule are adhered to, the goal of the tester is to test as much as possible for the longest possible time. The architect aims to build a “technically solid” and “clean” system, and the developer would like to implement the required functions as quickly as possible. This can result in conflicts of interest between the people involved in the project. They must be continuously negotiated and resolved in every phase of software engineering. Importantly, giving one individual person important competing roles like project manager and quality manager does not solve such conflicts. Instead, this approach harms state-of-the-art paradigms, such as separation of duty.

The quality of the results of the individual activities is therefore heavily dependent on close interaction and communication between the people involved. In particular, the personal relationships between these people and the interactions between the tasks assigned to the roles are decisive factors that contribute to the success or failure of a software project.

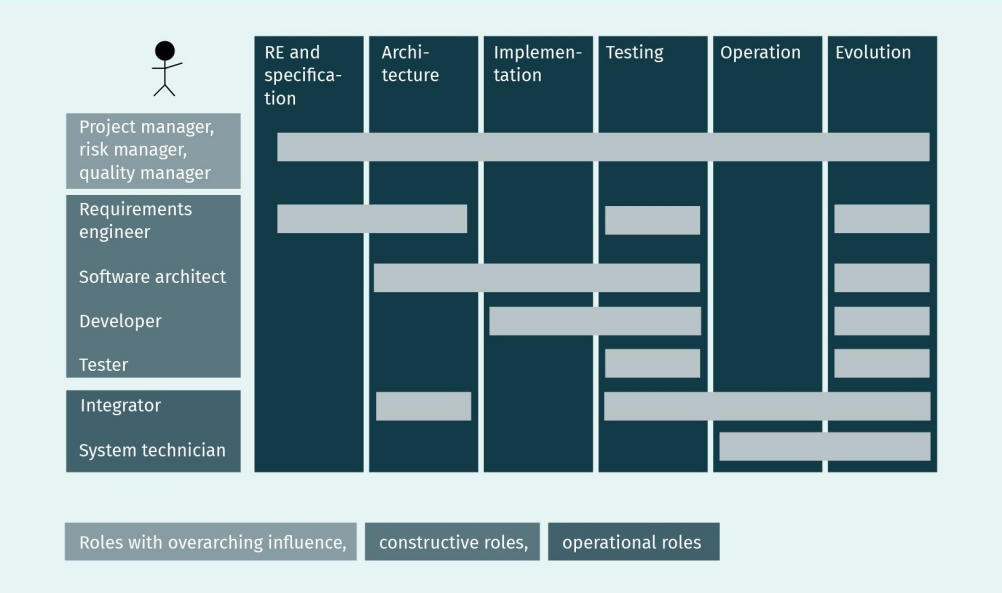
A basic assumption behind all considerations in software engineering can be described as follows: Projects work and can be completed successfully if the right people work properly with each other using the appropriate methods, (modeling and programming) languages, and tools. Ensuring these constraints is therefore an essential task in software engineering.

7.2 Typical Roles

Since there is no complete and generally accepted list of roles in software engineering, a selection of typical widely used roles is presented below. Both the specific name of their roles and the tasks, goals, and responsibilities vary from organization to organization, and often from project to project. There can also be specializations within a role, for example, sub-project manager, Java developer, or security architect.

The following table shows some common roles that are often found in practice. Their involvement in the software engineering life cycle is assigned using the columns. We will briefly explain each role.

Figure 18: Overview of Participation in Software Engineering According to Roles



Source: Created on behalf of IU (2023).

Project Manager

Project manager
This is the person assigned by the performing organization to lead the team that is responsible for achieving the project objectives.

The aim of the **project manager** is the successful implementation of the project. The tasks of the project manager include:

- planning, organizing, and estimating the cost of the project
- controlling project activities and people
- controlling the results achieved
- monitoring the project plan and productivity

They are active in all core activities across the board. The project manager creates plans for most of the other project roles, monitors compliance, and makes key decisions for the project.

Risk Manager

The aim of the risk manager is to minimize the overall risk of the project. Typical tasks of a risk manager are as follows:

- identifying risks of all kinds in the project
- analyzing and assessing risks
- developing strategies for risk resolution or risk mitigation
- monitoring risks

Like the project manager, the risk manager is active in all core project activities, often as part of project management. With the assessment of risks and the development of strategies to minimize them, the risk manager assumes an overriding role in the project.

Quality Manager

The quality manager is responsible for all activities and deliverables for quality assurance. They coordinate measures to ensure process and system quality. Specific activities of the quality manager are as follows:

- defining of system quality
- creating of a verifiable plan for quality assurance
- coordinating of quality assurance measures
- observing and ensuring customer satisfaction
- adopting of quality guidelines and checking whether they are compliant

The quality manager is active in all phases of the project, receiving specifications from and reporting to the project manager.

Requirements Engineer

The requirements engineer is responsible for delivering the business needs of the project, i.e., a set of requirements for the software system that have been coordinated with all relevant people. Specific activities of the requirements engineer are as follows:

- determining relevant sources for system requirements
- requirements elicitation
- documenting of the identified requirements
- achieving compliance with the documented requirements
- validating and coordinating the documented requirements
- tracking the requirement's life cycle

The requirements engineer is particularly involved in activities that determine the properties of the system in development. They receive specifications and follow-up from project management and quality management.

Software Architect

The aim of the software architect (or software designer) is the successful technical implementation of the business requirements in a software system. The activities of architects are as follows:

- gathering and understanding the domain-specific, technical, and organizational requirements for the system in development
- designing and describing of the (software) architecture of the system
- ensuring compliance with quality requirements
- ensuring compliance with technical boundary conditions
- monitoring the architecture during implementation
- preparing the system integration through documentation of the system architecture and definition of test cases for the integration test

The architect designs the architecture based on the results of the requirements engineer, taking into account additional requirements from the roles responsible for implementation, test, operation, and evolution. The architect must also adhere to the requirements of the project and quality management.

Of all the roles in software engineering, the architect is the most diverse in practice. In addition to various architectural levels for which an architect can be responsible, there are sometimes very different responsibilities and tasks of architects depending on the organization and project.

Developer

The developer's goal is to create program code that can be compiled into an executable system and comply with the specification. The main activities necessary for this are as follows:

- programming individual modules (units) of the system
- documenting the generated program code
- defining of the module-specific algorithms
- performing tests at unit level
- integrating individual modules into an overall system

The program code is created on the basis of the specification and the architecture. In this way, the developer implements the specifications made by the specifier and architect in a concrete system. To do this, the developer must adhere to the requirements of project and quality management, compliance with which is also checked by project and quality management. Compliance with the technical specifications is monitored by the architect, and compliance with the requirements is checked by the tester.

Tester

The aim of the tester is to find defects in the parts of the system that are completed in the course of the software project, and defects in the system as a whole. The specific activities of a tester are as follows:

- selecting of the test procedures to be used
- detailing of test plan, test cases, test data, and test run specification
- performing tests at various test levels
- documenting of the test conditions and results

The tester is responsible for the most complex quality assurance measures: testing the created software. Based on the results of the requirements engineer and architect, they check the results of the developer. To do this, the tester must adhere to the requirements of project management and quality management.

Integrator (Transition Manager)

The aim of the integrator is to adapt the newly created or changed system to the existing landscape of software and hardware. Typical activities of the integrator are as follows:

- examining the integrability of the system in individual components of the operating infrastructure
- trying out selected aspects of integration
- forecasting the runtime behavior and hardware requirements of the new system in the operating infrastructure
- integrating of the finished system into the operating infrastructure
- establishing the connection of technical interfaces between systems

The integrator incorporates the system supplied by the developers into the existing system landscape. To do this, they must adhere to the requirements of project management and quality management, by which they are also checked.

System Technician or Manager

The aim of the system technician (system manager) is to provide and guarantee technical resources for the operation of a software system. In the context of virtualization or containerization, the technical resources of the assigned virtual machines or containers must also be managed by such a role, though the task may be outsourced. The activities of a system technician are as follows:

- evaluating, installing, and maintaining the operating infrastructure (hardware and software)
- providing the resources required by individual software systems
- monitoring the available and used resources, such as hard disk space, main memory, central processing unit (CPU) time, and network utilization
- establishing, operating, and maintaining systems to ensure the availability of the operating infrastructure

Although the system engineer is not involved in creating a software system, the software architecture of the system must match the operating infrastructure. After development and integration have been completed, the system technician ensures proper operation according to specifications from project management and quality management.

SUMMARY

In software engineering, a distinction is made between different roles, each of which is characterized by its name, tasks, and objectives. One person can take on several roles, and several people can be assigned the same role. The division into different roles also induces conflicting goals between individual roles, which are continuously negotiated and resolved between the people involved in the project.

Typical roles with an overarching sphere of influence are project managers, risk managers, and quality managers. These roles create plans for the other roles and check that they are adhered to. Typical roles directly involved in software creation activities are requirements engineer, software architect, developer, and tester. Following the creation, the integrator and system technician roles are responsible for the integration and operation of the system.

UNIT 8

ORGANIZATION OF SOFTWARE PROJECTS

STUDY GOALS

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

- describe the relationship between process paradigms, software process model frameworks, software process models, and software processes.
- explain the basic principles of the waterfall model, V-model, and evolutionary (Agile) development.

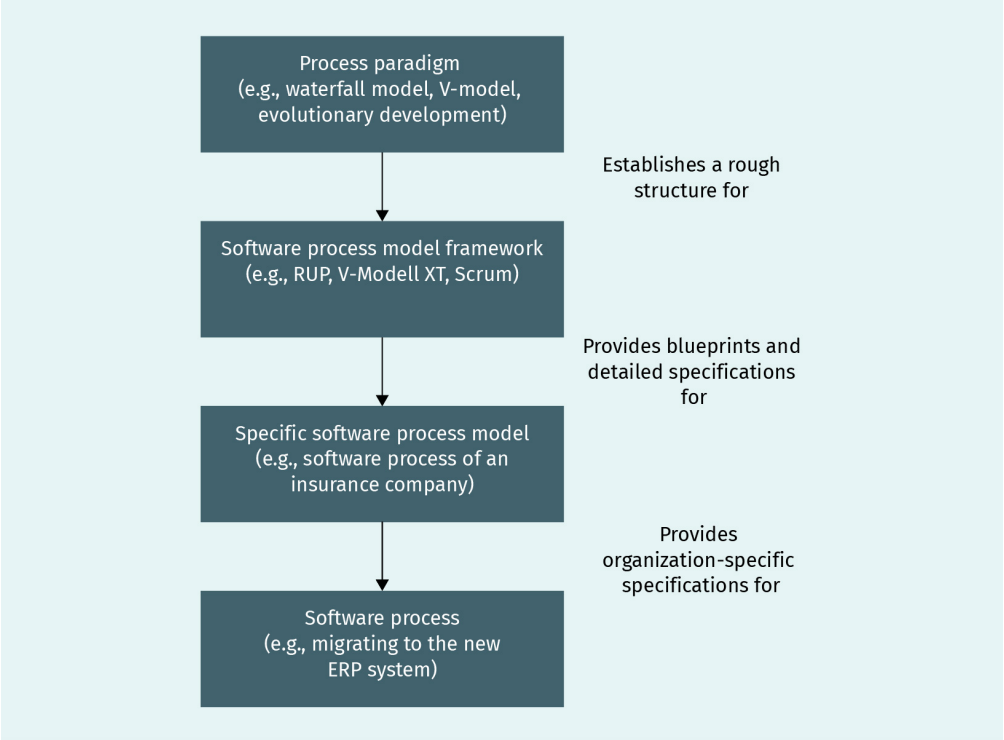
8. ORGANIZATION OF SOFTWARE PROJECTS

Introduction

Due to the organizational complexity of enterprise software development projects that arise from the involvement of the various roles, as well as the diverse activities and their dependencies, software processes cannot be organized ad hoc or chaotically. When carrying out a specific software process, project teams can be guided, in practice, by company-specific software process models. These, in turn, are derived from frameworks, which follow certain paradigms. The figure below illustrates the relationship between process paradigm, software process model framework, individual software process model, and software process described below. We will examine the terms shown and their relationships in detail in this unit. We will also assess the relevance of these concepts in practice.

In the literature, as well as in practice, the terms “software process,” “process model,” and “process paradigm” are not used uniformly. In particular, the terms “process model” and “software process” are often used to denote process paradigms, process frameworks, or individual process models, depending on the situation. It is good practice to clarify these terms among the team at the beginning of a project.

Figure 19: From Process Paradigm to Software Process



Source: Created on behalf of IU (2023).

8.1 From Process Paradigm toward Software Process

Process paradigms are very general process models. They provide the rough structure of software processes. The process paradigm defines the basic principle of a software process model without explaining specific roles, responsibilities, or detailed processes. Examples of process paradigms are the waterfall model, V-model, and evolutionary development.

Process paradigm
This defines basic principles that provide a rough structure for designing processes.

Each information technology (IT) organization usually has its own individual specifications and constraints for activities in software engineering. Therefore, software process models that have already been created cannot be fully transferred to other organizations, but must always be adapted. For this reason, detailed **software process model frameworks** were created as a basis for the design of organization-specific software processes. These frameworks serve as a template when creating a company-specific software process model. A software process model framework follows the structure specified in the process paradigm and expands it with detailed descriptions of roles, responsibilities, concrete activities, and their dependencies. Examples of widely used software process model

Software process model framework
This is a comprehensive and detailed set of general roles, activities, and responsibilities that can be tailored to the specific needs of IT organizations.

frameworks are V-Modell XT, Rational Unified Process (RUP), and Scrum. Many organization-specific software process models are derived from them and adapted to individual requirements or specific projects.

To take into account the organization-specific features in software development, specific software process models are developed based on the specifications of the software process model frameworks. On the basis of this individual process model, the way a development process should run is described. An individual software process model defines the following:

- which activities are to be carried out during software development
- the order in which the activities are to be carried out
- the roles and organizational units that are responsible for carrying out the activities
- the tools and methods to support these activities
- the types of objects that will be created and processed by performing the activities

A single software process, i.e., a very specific software project, is an instantiation of the individual software process model. For a software process model, there are a number of software processes that are executed according to the specifications and framework conditions of the process model. Based on the specifications of the process model, the following must be specified for a process or specific project:

- which activity is carried out by whom and when
- which objects are processed or generated when an activity is carried out

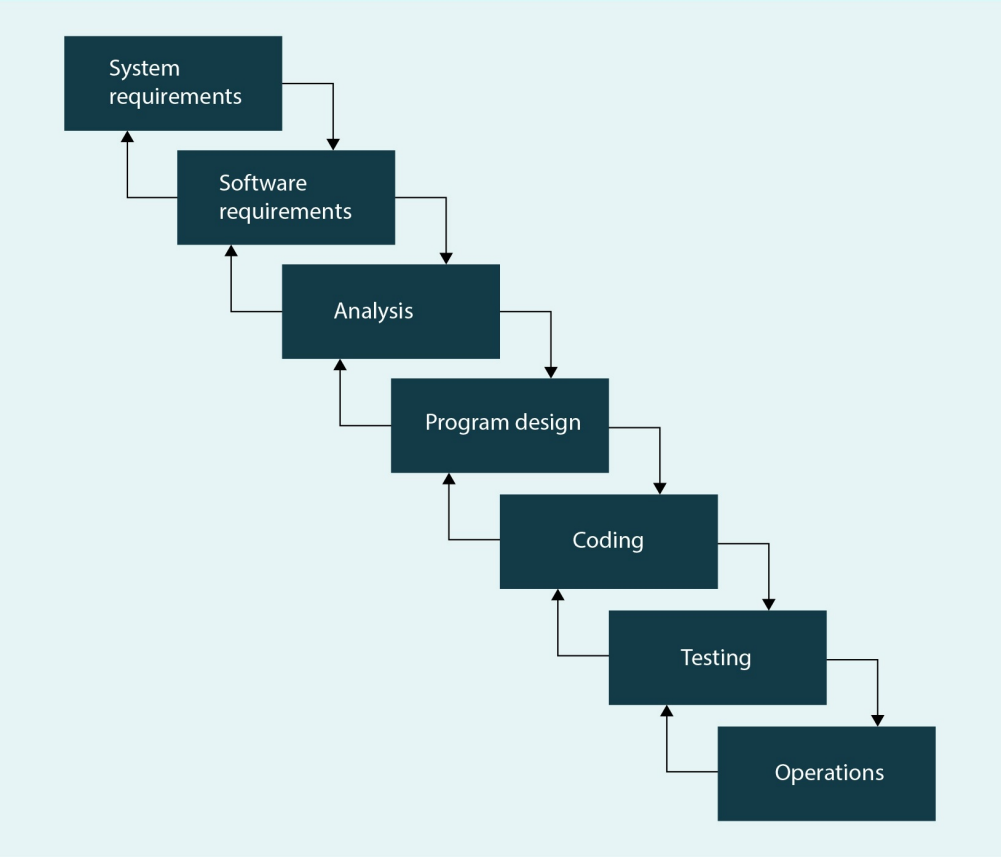
8.2 Process Paradigms

The general process models in software engineering identified by the term “process paradigm” are based on findings and observations made in the course of the implementation of enterprise software projects. As these findings evolve, the paradigms also evolve. New paradigms (the Agile approach) are usually perceived as extreme and followed rigidly, then applied excessively during hype, and finally mature with increasing experience.

Waterfall Model

The waterfall model shown in the figure below was described by Royce (1970) and is the oldest process paradigm. The basic idea behind the waterfall model is the step-by-step processing of the individual phases of requirements, analysis, design, implementation, testing, and operation in a defined sequence. Like a waterfall, a software project “falls” one after the other from “top to bottom.” Each individual phase is fully completed, and only then does it move on to the next phase. There is the possibility of feedback between adjacent phases. If errors are found or there are problems with the result of a phase in the subsequent phase, the software process can go back one phase. This way, neighboring phases can alternate several times in a row in the event of a problem.

Figure 20: Waterfall Model



Source: Benner-Wickner (2021), based on Royce (1970).

With the waterfall model, the software projects, which were often unstructured, are given a fixed structure. In addition to the strict separation of the individual phases, it attaches great importance to complete and up-to-date documentation. A phase transition can only take place if the results of a phase are fully and extensively documented, and the quality has been successfully assured.

From a management perspective, the waterfall model has the advantage that the software process is divided into clear phases with defined results. Thus, clear organizational responsibilities can be defined, and the current status and project progress can be monitored. The task areas can be clearly structured, especially in distributed projects in which several organizations work together at several locations. By demanding comprehensive and complete documentation of the results, transfer points can be contractually agreed between individual organizations.

However, the definition of a rigid sequence of activities is the opposite of the knowledge-driven nature of software projects. Although it is often requested, the requirements for industrial information systems cannot be fully described at the beginning of a software process. All further activities up to implementation are used to clarify requirements that are either unknown or not yet clear. In addition, strict adherence to the waterfall model

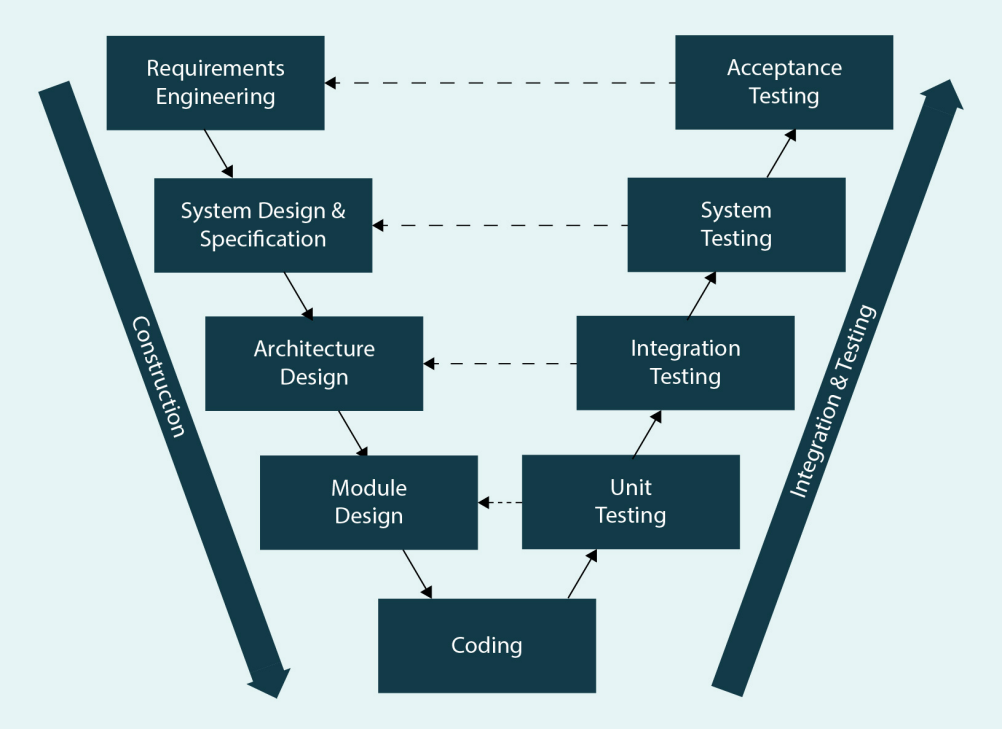
prevents the start of successor phases for individual elements that have already been completed. Even if, for example, a large part, but not the entire system, is specified, the architecture or implementation must not begin yet, according to the waterfall model. Furthermore, the quality assurance measures for the fully documented results of the individual phases ensure that one has to wait for the test to be completed between the completion of the result and further work.

From today's perspective, the waterfall model should only be used as a rough guide when creating software process models for industrial information systems. In fact, shortly after Royce's publication, his model was supplemented with iterations. Nevertheless, the requirement for clearly separated phases and the focus on documentation that is complete at all times often leads to large documents being created and checked with a great deal of effort, then quickly becoming obsolete.

V-Model

The process paradigm "V-model" was developed toward the end of the 1970s by Boehm, and is a basis for various software process models in the public sector (Boehm, 1979). The basic idea of the V-model, as shown in the figure below, is the 1:1 assignment of construction phases (the left half of the V) and phases to be tested (the right half of the V).

Figure 21: V-Model



Source: Markus Kleffmann, 2023.

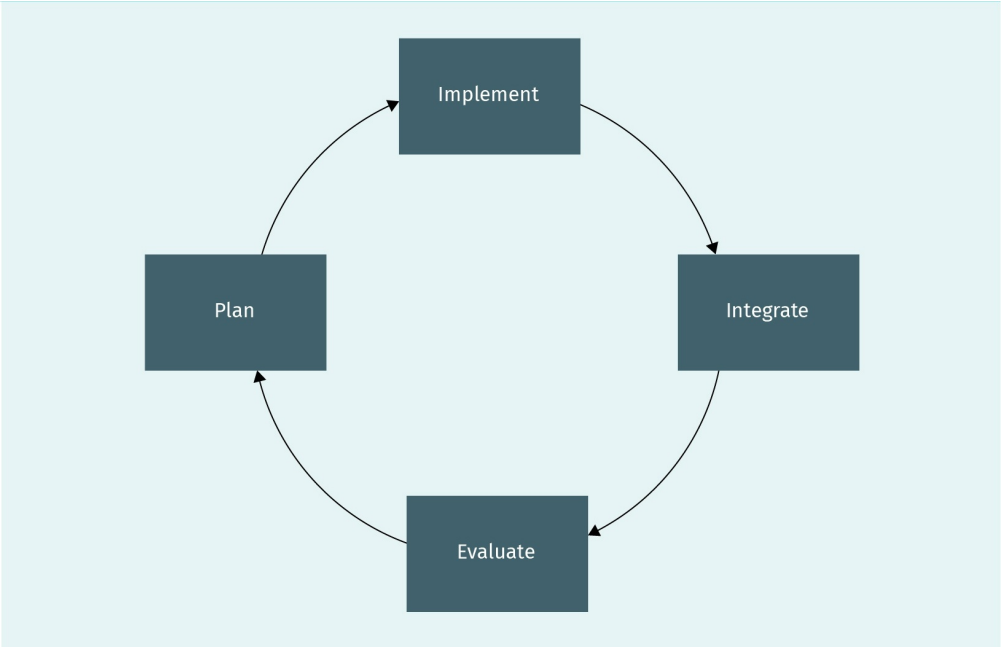
The path through the software process is a result of the sequence of the defined phases. Starting with the technical requirements analysis, as in the waterfall model, all of the core software engineering activities are carried out. The software project then moves through various test levels up to the acceptance of the system. Each phase of the construction is assigned a specific test level, which is on the same level of detail as the construction phase. As with the waterfall model, it also applies to the V-model that the previous phase must be completed successfully. Before the integration tests can begin, all unit tests must be completed.

Evolutionary and Agile Development

The process paradigm “evolutionary development” (also known as Agile development) describes a general process model. Its basic idea is the creation of the software system in several repetitive cycles (also known as sprints in the context of the Scrum framework). The functional scope of the software system in development grows with each version. This iterative approach goes back to the spiral model by Boehm (1988). Findings that are gained during the implementation and evaluation of the current version can be considered in following system versions. The figure below illustrates a general scheme of evolutionary development. Instead of focusing on complete documents and specifications, the focus here is on a version of executable software. A cycle runs as follows:

1. Determine which functions are to be implemented in this cycle.
2. Implement the functions.
3. Integrate the new functions into the existing system.
4. Test and evaluate the current software version.

Figure 22: Cycle of Evolution



Source: Created on behalf of IU (2023).

In contrast to the waterfall model, the software process in evolutionary development does not fall strictly “from top to bottom” along the core activities of software engineering. In evolutionary development, the software process “turns in a circle” around a finished piece of software. The software engineering core activities are not carried out and documented in full in one phase, but in smaller parts that interlink in each cycle.

This way, evolutionary development considers that many insights into requirements and technological solutions can only be gained during software development. The new knowledge can be incorporated directly in each new cycle. In addition, errors and inaccuracies in the specification or the program code can be quickly identified and eliminated. Another advantage of evolutionary development is the rapid availability of a basically executable system; even if not all functions have been fully implemented, there is already a result that can be used in parts.

Compared to the waterfall model, there are no phases with clearly defined results, but all software engineering core activities are repeated in small parts in short cycles. For this reason, no complete specification is created at the beginning of the software process. This means that the specific functional scope of the system is only precisely defined during development. Therefore, from a management perspective, the evolutionary development appears unstructured and without a clearly defined result. With the main focus on creating an executable software system, there is the risk of generating program code quickly and neglecting the necessary evolution of the system architecture, the accompanying architecture, and system documentation.

Currently, the term “agility” or “Agile software development” does not refer to specific techniques or processes, but rather a thought scheme or a design principle for software development activities. There is no uniform definition for this. Often, “agility” is also seen as a value or attitude, according to which one designs daily project work. In contrast, the “classic” way of thinking in software engineering is often referred to as “plan-driven.”

A fundamental difference between the two approaches is the basis on which important decisions are made in the project. Agile software development is geared towards ensuring that all important decisions are made based on knowledge gained during the project. So, this approach is “knowledge-driven.” Classic software development, on the other hand, focuses more on plans that were created in the run-up to the project or in the early project phases. Based on previous experience and the current level of knowledge of the people involved, assumptions about the course and results of the project are made and documented in the form of plans and specifications. Thus, these projects are “assumption-driven” because they were created based on assumptions about the future (Rubin, 2013).

SUMMARY

In enterprise software engineering, the software development process is structured by specifying software process models. Process paradigms are very general process models. They provide the rough structure of software processes without going into detail. A software process model

framework follows the structure specified in the process paradigm and adds detailed descriptions of roles, responsibilities, specific activities, and their dependencies. A software process model framework serves as a template for creating our own organization-specific software process model. This individual process model describes how a development process should run transparently for all those involved in an organization.

The basic idea behind the process paradigm waterfall model is the step-by-step processing of the individual phases of requirements, analysis, design, implementation, test, and operation in a defined sequence. Like a waterfall, a software project “falls” one after the other from “top to bottom.” In the V-model, each phase of the construction is assigned a specific test level. As in the waterfall model, the path through the software process results from the sequence of the defined phases. The idea behind the process paradigm “evolutionary development” is to construct a software system in several repetitive cycles. The software engineering core activities are not completed in one phase, but in smaller parts that interlink in each cycle.

UNIT 9

SOFTWARE PROCESS MODEL FRAMEWORKS

STUDY GOALS

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

- name the main elements of the V-Modell XT and explain how they are used.
- describe the Rational Unified Process (RUP) is and its phases.
- understand which roles and activities are included in Scrum.

9. SOFTWARE PROCESS MODEL FRAMEWORKS

Introduction

The software process model frameworks V-Modell XT, Rational Unified Process (RUP), and Scrum presented in this unit are widely used in practice. Many organization-specific software process models are derived from them and adapted to individual requirements or specific projects.

With the V-Modell XT, a framework for software process models was created that is mandatory for many public information technology (IT) projects, for example, in the German federal administration. This means that the V-Modell XT is very popular in the public sector. The Rational Unified Process (RUP) is an industrial framework for software process models that was developed by Grady Booch at Rational Software (acquired by IBM in 2003) (Bednarz, 2002). Scrum is an evolutionary software process model framework that can be characterized by short cycles and a self-organizing team. Interestingly, Scrum does not define roles or activities specific to software engineering, such as an IT architect or programmer. Strictly speaking, Scrum is not a software process model framework, but can be used as a framework for process models for very different tasks.

9.1 V-Modell XT

The main elements of the V-Modell XT framework are

- project types,
- decision points and project implementation strategies,
- process modules, and
- references.

On the basis of these elements, we can, in principle, design and adapt our own software process models. As a result, V-Modell XT is much less rigid than the V-model.

Project Types

In order to enable the use of the V-Modell XT in as many projects as possible, a distinction is made between the following different project types (Weit e.V., 2006):

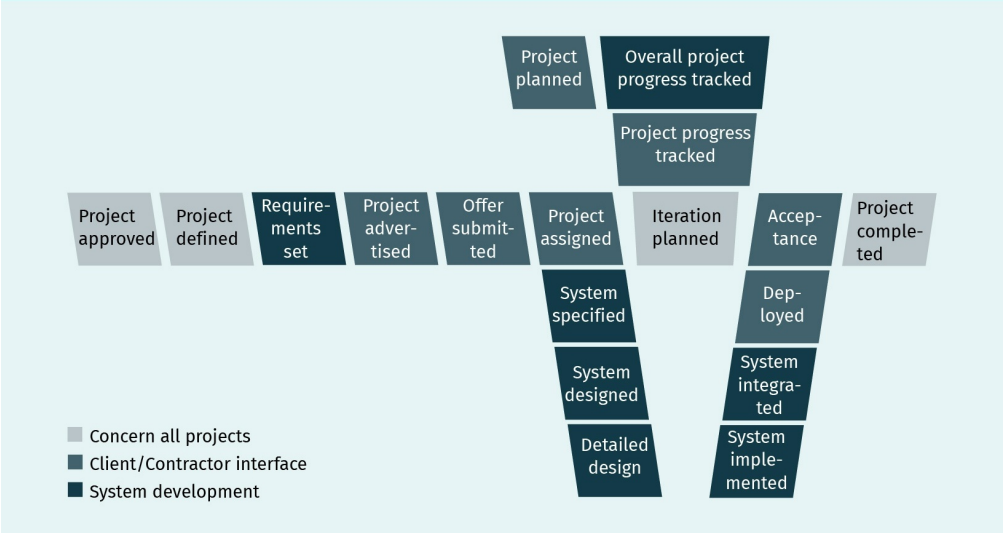
- system development project of a client
- system development project of a contractor
- system development project of a client with a contractor in the same organization.

For each of these project types, the V-Modell XT describes which process modules must be used and which can be optionally selected.

Decision Points and Project Implementation Strategies

While considering decision points in the V-Modell XT, it is decided whether a certain project progress level is reached. Which results or products will be completed there is defined for each decision point. The order of the decision points is determined in the project implementation strategy. The concrete implementation strategy for a project is derived from the project type and the specific project features. The figure below contains an overview of the decision points relevant in the V-Modell XT and their dependency on the project type.

Figure 23: Decision Points in V-Modell XT



Source: Benner-Wickner (2021), based on Weit e.V. (2006).

Process Module

In the process modules in the V-Modell XT, specific tasks, activities, results, and the roles involved are defined within a software project. As with the decision points, the specific project type also decides with the process modules whether they must be used or not. In the V-Modell XT, a process module specifies what is to be done and by whom in a project.

References

As a basis for the organization and project-specific adaptation, the V-Modell XT, with its extensive references, provides specifications and guidelines for the following:

- tailoring for your own software process models
- roles in the software process (over 30 different roles with a description, responsibilities, and category)

- products in the software process, whereby the V-Modell XT contains a detailed description of the creation, use, and dependencies of 110 product or result types
- activities, whereby detailed instructions for creating and processing the results are given for each of the more than 100 different activities with the course of the individual work steps
- convention mappings in which national and international conventions are related to elements of the V-Modell XT

These serve as a reference work, both for the construction of your own software process models, and for the implementation of a software project.

9.2 Rational Unified Process (RUP)

Rational Unified Process
This is a framework for software process models that is widely used in the software industry around the world.

According to Kruchten (2003), the **Rational Unified Process** (RUP) divides the software process into four phases:

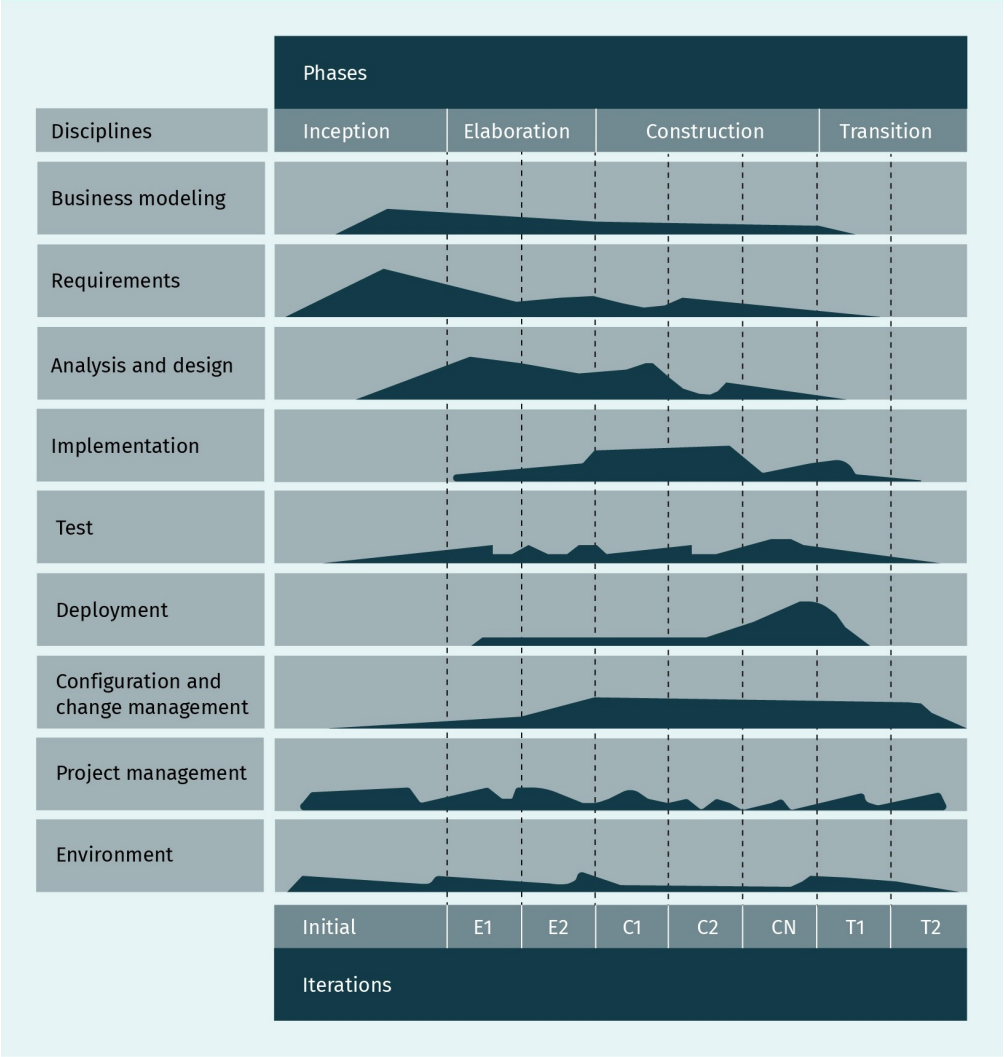
1. Inception
2. Elaboration
3. Construction
4. Transition

In the inception phase (the preparation of the project), the focus is on requirements engineering, developing the project vision, initial risk analysis, creating the project plan, and drawing up the project budget. In the elaboration phase, the requirements are refined and detailed, a concept for the system architecture is developed and, described and, if necessary, the construction of the first prototypes is started. An important result of this phase is the architecture description, which serves as a specification and framework for the following construction phase. Most of the implementation activities take place in construction. In this phase, the program code of the software system is mainly generated. In addition, software tests are carried out and corrected. The transition phase includes all activities that must be carried out after the development work has been completed through to commissioning. Among other things, the user training, the system test, the integration into the execution environment, and the documentation all fall into this phase.

Organization of the Software Engineering Core Activities

The left column in the figure below contains the software engineering core activities, referred to as disciplines in the RUP. For each discipline, the RUP names specific roles, artifacts (results, products), activities (work steps, tasks), and workflows (sequence of activities in connection with artifacts and roles). In contrast to the waterfall model, the software engineering core activities of a phase are not strictly separated, but sometimes take place in parallel or are closely linked. However, the phases in the RUP differ in the proportion or intensity with which the activities are carried out. This is shown in the figure below with the different effort curves in the rows.

Figure 24: RUP Disciplines and Phases



Source: Brückmann (2021), based on Kruchten (2003).

The Rational Unified Process supports the evolutionary development through the possibility of executing the individual phases several times, one after the other (iterations). This is illustrated by the last line of the figure: E1 and E2 are two iterations cycles of the elaboration phase, C1, C2, and CN are cycles of the construction phase, and T1 and T2 are cycles of the transition phase. The number of cycles and whether they are carried out depends on the specific software project.

9.3 Scrum

Scrum defines various roles, specifies strict requirements for activities and their order, and introduces Scrum-specific elements for managing the project (Rubin, 2013).

Roles

Product Owner
The Product Owner represents the customer and is responsible for the success of the product.

There are three central roles in Scrum: the **Product Owner**, the Scrum Master, and the team (Rubin, 2013).

The Product Owner represents the needs of the customer or the user. In Scrum, they are responsible for the success of the project, as well as for the requirements engineering. The Product Owner forms the bridge between the project and the outside world because they are the only interface between stakeholders and developers of the system. Important technical decisions in Scrum are made by the Product Owner. They create the release plan, prioritize the requirements, and approve the result created in a sprint (Rubin, 2013).

The team is responsible for the technical conception, construction, and quality assurance of the software. A team works in a self-organized and autonomous manner. In coordination with the Product Owner, the team determines how many functions are implemented and when they are implemented in a sprint. Apart from organizational requirements and conventions, the team is free to choose the technical implementation. All roles required to implement a software system, for example, architect, developer, and tester, work closely together in a team of between three and nine people. At the end of a sprint, the team presents their result (Rubin 2013).

The Scrum Master is comparable to an organizational project manager. They moderate and accompany the activities in Scrum and ensure that the time and organizational requirements in a Scrum cycle are adhered to by all those involved. In addition, they ensure the ideal working environment for the Product Owner and team by removing the obstacles that hinder efficient work. The Scrum Master has no authority to issue instructions to the Product Owner or the team. Instead, they manage the Scrum process by enabling the development team and the Product Owner to work. This is done by protecting them from external influences and disruptions, and in solving organizational issues (Rubin, 2013).

Elements for Managing and Controlling the Project

The most important concepts that are used in a Scrum process are the product backlog, the sprint backlog, and the velocity.

The product backlog is a list of all system requirements. The detailing of the requirements ranges from an initial idea to a fully specified function. At the beginning of a software project, the product backlog often only contains a few, roughly outlined requirements and wishes for the system. During the process, the requirements are refined, supplemented, and prioritized by the Product Owner. Only the Product Owner can make changes to the product backlog (Rubin, 2013).

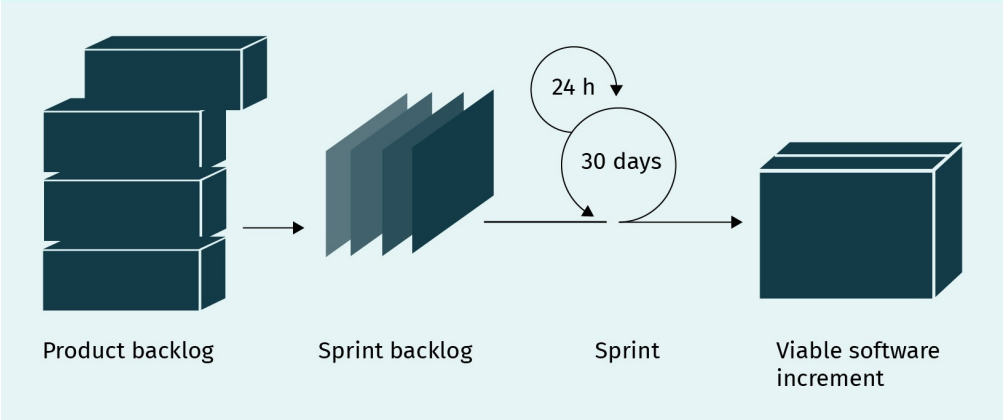
The sprint backlog is a list of requirements that are to be implemented by the team in a sprint (this is a cycle that uses evolutionary development). The sprint backlog is a subset of the product backlog. The Product Owner and team agree on which requirements are included in the sprint backlog. The number of requirements depends on the team's velocity. The sprint backlog is not changed during the processing time of a cycle (Rubin, 2013).

The velocity of a team is a team-specific key figure that results from the amount and scope of the functions implemented by a team in a cycle. Typically, the velocity of a team stabilizes with experience after the first five to seven cycles performed by the team. The velocity provides an indication of how many and which requirements a team can load from the product backlog into a sprint backlog for the duration of a cycle (Rubin, 2013). Please keep in mind that it is bad practice to compare Scrum teams by their velocity, as this value is very team-specific.

Scrum Activities

The general structure of Scrum is shown in the figure below. The sprint backlog is created from the product backlog for each sprint (this is the name of a cycle in Scrum). The team then implements the requirements from the sprint backlog during a sprint, which can last, for example, 30 days. There is a short status meeting every day in which each team member briefly presents what they have done in the past 24 hours, what problems they have, and what they will do in the next 24 hours. The daily meeting is called "daily Scrum" or "stand-up meeting," and can last a maximum of 15 minutes. At the end of a sprint, the team presents the software, which has been expanded to include additional functions, and determines the list of requirements for the next cycle (Rubin, 2013).

Figure 25: Scrum



Source: Benner-Wickner (2021), based on Rubin (2013).

Course of a Cycle (Sprints)

In Scrum, which individual phases a cycle consists of and which roles carry out which activities are precisely defined. A sprint can be divided into four different phases:

1. Sprint planning
2. Sprint
3. Sprint review
4. Sprint retrospective

In the sprint planning phase, the Product Owner presents the requirements that they have planned for the current sprint. The team estimates the implementation effort for each of the requirements, and the sprint backlog is filled. To do this, the team selects the requirements that are prioritized by the Product Owner so that the total effort of all requirements in the sprint backlog corresponds to the team's velocity. The team then plans how the requirements will be implemented, and who is responsible for which requirement (Rubin, 2013).

During the sprint, the team implements the requirements from the sprint backlog. The Product Owner is available to answer any questions. Mainly, the Product Owner specifies the requirements for the next sprint during implementation. The set of requirements in the sprint backlog is not changed during a sprint. The Scrum Master removes obstacles that arise during the implementation, and ensures that the daily stand-up meeting is carried out properly (Rubin, 2013).

In the sprint review, the team presents its results to the stakeholders and the stakeholders give their feedback on the result. The Product Owner officially accepts the sprint result and documents changes and requirements that have not yet been implemented in the product backlog (Rubin, 2013).

The last phase in the sprint is a sprint retrospective. The Product Owner, team, and Scrum Master reflect on their own sprint from the points of view of "what went well?" and "what needs to be improved?" (Rubin, 2013).

SUMMARY

The V-Modell XT was created as a framework for software process models in governmental IT projects. The main elements of the V-Modell XT framework are project types, decision points, project implementation strategies, process modules, and references.

The Rational Unified Process (RUP) is an industrial framework for software process models. In the RUP, the software process is divided into four phases: inception, elaboration, construction, and transition.

Scrum is an evolutionary process model framework that can be characterized by consistent organization in short cycles and a self-organizing team. To this end, Scrum defines specific roles, specifies strict requirements for activities and their order, and introduces Scrum-specific elements for managing and controlling projects.

On the basis of these frameworks for software process models, organization-specific software process models are created, which, in turn, serve as specifications for individual software development projects.

BACKMATTER

LIST OF REFERENCES

Agutter, C. (2019). *ITIL® foundation essentials: The ultimate revision guide* (4th ed.). IT Governance Publishing.

The Association for Computing Machinery. (n.d.). *ACM proceedings*. https://dl.acm.org/proceedings

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

Bednarz, A. (2002). *IBM shells out $2.1 billion for Rational Software*. Network World. https://www.networkworld.com/article/2338780/ibm-shells-out--2-1-billion-for-rational-software.html

Boehm, B. W. (1979). *Software engineering: R & D trends and defense needs.Research directions in software technology*. MIT Press.

Boehm, B. W. (1988). A spiral model of software development and enhancement. *IEEE Computer*, *21*(5), 61—72. https://doi.org/10.1109/2.59

The Centre of Computing History. (n.d.). *John von Neumann*. http://www.computinghistory.org.uk/det/3665/John-von-Neumann/

Cohane, R. (2017). *Has the software crisis passed?* Medium. https://medium.com/@ryancohane/has-the-software-crisis-passed-d45ce975a1e7

Crawford, J. (n.d.). *Wireframe for tyler.petersonrace.com* [Image]. Flickr. https://www.flickr.com/photos/37996599088@N01/4424880628

The Editors of Encyclopaedia Britannica. (n.d.). George Boole: British mathematician. Encyclopaedia Britannica. https://www.britannica.com/biography/George-Boole

FileFormat.info. (n.d.). *Complete character list for UTF-8*. https://www.fileformat.info/info/charset/UTF-8/list.htm

Glinz, M. (2017). *A glossary of requirements engineering terminology*. International Requirements Engineering Board. https://www.ireb.org/content/downloads/1-cpre-glossary/ireb_cpre_glossary_17.pdf

Glinz, M., van Loenhoud, H., Staal, S., & Bühne, S. (2020). *Handbook for the CPRE foundation level according to the IREB Standard*. International Requirements Engineering Board. https://www.ireb.org/content/downloads/5-cpre-foundation-level-handbook/cpre_foundationlevel_handbook_en_v1.0.pdf

The 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

The International Organization for Standardization. (2017). *Systems and software engineering—Software life cycle processes* (ISO Standard No. 12207:2017). https://www.iso.org/standard/63712.html

The International Organization for Standardization, & The Institute of Electrical Engineers, Inc. (2017). *Systems and software engineering—Vocabulary* (2nd ed.). https://standards.iso.org/ittf/PubliclyAvailableStandards/c071952_ISO_IEC_IEEE_24765_2017.zip

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

Jones, K. (2019). *The big five: Largest acquisitions by tech company*. Visual Capitalist. https://www.visualcapitalist.com/the-big-five-largest-acquisitions-by-tech-company/

Kruchten, P. (2003). *The rational unified process: An introduction*. Addison-Wesley.

Leach, D. P., Malvino, A. P., & Saha, G. (2011). *Digital principles and applications* (7th ed.). McGraw-Hill.

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*. https://en.wikipedia.org/wiki/Binary_file

Nozero. (2015). *Configuring router with integrated ADSL modem running PPPoE*. DSL Reports. https://www.dslreports.com/faq/8193

Risska, V. (2010). *Spezifikationen für BDXL mit 128 GByte veröffentlicht* [Specifications for BDXL with 128 GByte published]. Computer Base. https://www.computerbase.de/2010-06/spezifikationen-fuer-bdxl-mit-128-gbyte-veroeffentlicht/

Royce, W. W. (1970). *Managing the development of large software systems: Concepts and techniques*. IEEE WESCONN.

Rubin, K. S. (2013). *Essential Scrum*. Addison-Wesley.

Shipley, C., & Jodis, S. (2003). Programming languages classification. In H. Bidgoli (Ed.), *Encyclopedia of information systems* (pp. 545—552). https://doi.org/10.1016/B0-12-227240-4/00138-6

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

Tanenbaum, A. S. (2013). *Structured computer organization* (6th ed.). Pearson.

The Importance of Being Earnest. (n.d.). *TIOBE index for June 2021*. http://www.tiobe.com/tiobe-index?20210605

Weit e.V. (2006). *V-Modell XT*. http://ftp.tu-clausthal.de/pub/institute/informatik/v-modell-xt/Releases/2.3/V-Modell-XT-Gesamt.pdf

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.