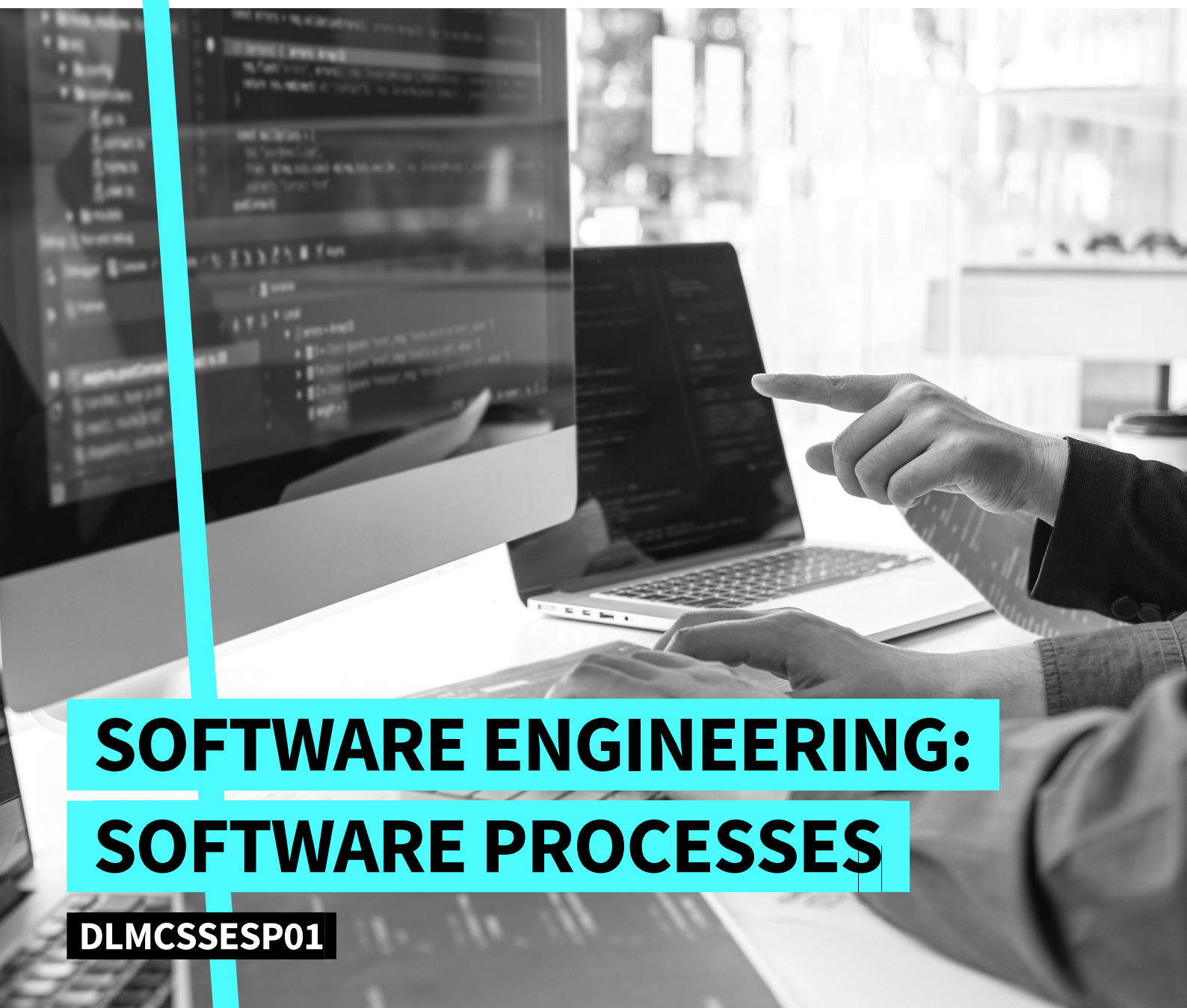


Course Book



SOFTWARE ENGINEERING: SOFTWARE PROCESSES

DLMCSSESP01

iu

INTERNATIONAL
UNIVERSITY OF
APPLIED SCIENCES

LEARNING OBJECTIVES

Software processes and life cycle models are applied in software engineering to facilitate methodical development of software and software-intensive systems. This course will introduce the foundations of software processes. Moreover, it will demonstrate software process definition and modeling.

In **Software Engineering: Software Processes**, you will discover the role of software processes and life cycle models in software engineering, from initialization to withdrawal of a software system. You will learn about project management in software engineering, and explore the origin, establishment, and evolution of software processes and models. Upon completion, you will have an understanding of modeling notations and meta-models. Furthermore, you will be able to apply applicable notations to define and model software processes, as well as interactions between processes.

Basic software life cycle models, **Agile**, and lean development processes are explained. You will gain an understanding of the advantages and shortcomings of plan-driven and Agile development, and the value of applying a hybrid approach. This course will introduce Scrum development, common Agile practices, and scaling of Agile development.

Furthermore, customizable process models will be explored. **Information technology (IT) service management and operations** will be introduced as a practice that ensures implementation and delivery of quality IT services for customers. You will learn about the culture of **DevOps**, and the management of safety, security, and privacy as it relates to software, data, and information.

This course will teach you about the management and governance of IT processes and services. You will learn about process design and deployment, as well as process tailoring. Furthermore, **it will introduce you to means whereby to** assess, measure, and improve processes, and **also tools to use support process modeling**, process management, and process enactment.

UNIT 1

FOUNDATIONS OF SOFTWARE PROCESSES

STUDY GOALS

On completion of this unit, you will have learned ...

- the role of software processes and life cycle models in software engineering.
- **about the** historical origins of processes and models.
- how processes and models have evolved throughout history.
- the typical challenges of managing information technology (IT) projects in software engineering.
- **about** project management in software engineering processes.

1. FOUNDATIONS OF SOFTWARE PROCESSES

Introduction

Software engineering has been an established field for quite some time. Many businesses and organizations are increasingly dependent on software and software-intensive systems to perform their core functions efficiently. They must therefore have access to suitable software that continues to function for their business requirements. Private individuals are also increasingly reliant on information technology (IT) and its associated software to ease, improve, and enrich their daily lives. Software malfunctions and failures can cause enormous economic damage or even physical harm. Thus, it is important to develop and implement quality software that is robust and **fit for its purpose**. Software must also be developed and maintained in such a manner that it adheres to acceptable levels of reliability, consistency, safety, security, usability, and privacy. Boehm (2006) defines software engineering as “the application of science and mathematics by which the properties of software are made useful to people” (p. 13).

Several different kinds of software engineering are available, and a number of viewpoints can be applied to explore software engineering. As an example, Boehm (2006) states that software engineering can be classified as “large or small; commodity or custom; embedded or user-intensive; greenfield or legacy/COTS/reuse-driven; homebrew, outsourced, or both; casual-use or mission-critical” (p. 12). The relevant viewpoint of the involved and affected stakeholders, the type of software engineering applied, the purpose of the envisaged software, and the nature of the software project dictate how software and software systems will be planned, designed, developed, implemented, and maintained. Regardless, it will still be by way of one or more pre-defined software processes and life cycle models that facilitate the methodical and structured design, development, implementation, and maintenance throughout its life cycle.

Continuous advancements in computer hardware and software, wider ranges of application and new insights gained by practitioners and researchers in terms of improved ways of executing development projects see to it that the field of software engineering is constantly evolving. IT projects have been (and still are) notoriously diverse and difficult to implement successfully in practice. **Various authors, such as Livari et al. (2000) and Clegg and Shaw (2008)**, argue that this is a result of development approaches that focus mostly on technological aspects of software, neglecting other relevant (e.g., human, social, and cultural) dimensions. Some projects fail **due to arising challenges**, but this leads to ongoing research and subsequent paradigm shifts in the field. The aim here is to streamline and evolve applied processes and models in order to manage projects more effectively. Kneuper (2018) **states that the issue is that** “software development is more similar to social science than natural science” (p. 3).

Accordingly, based on a variety of factors, such as the available technology at the time, the range of **application**, prevalent worldviews, prevailing research on future trends, and possible underlying causes of project failures in the past, a range of software engineering trends have emerged and evolved over the years. These trends include focusing on the engineering of computer hardware and algorithmic **(once-off)** programming in the 1950s, crafting (code-and-fix) as a programming approach of the 1960s, and attempts at formal and structured development methods in the 1970s (Booch, 2018; Boehm, 2006). Booch (2018) and Boehm (2006) note that increasing productivity and concurrent, risk-driven processing were the focus areas in the 1980s and 1990s respectively, agility and rapid development became essential in the 2000s, and the 2010s were characterized by global connectivity and integration. Currently, a multitude of different software engineering methodologies, software processes, and life cycle models exist (Kneuper, 2018). Increased awareness of underlying issues and attempts to manage the challenges and risks associated with software development, as well as technological advancements, bring about continuing research by academia and practitioners in the field to improve and refine software engineering processes and models. As a result, new trends continue to emerge, and processes and models continually evolve.

1.1 The Role of Software Processes

The term “process” originates from the Latin *processus*, which implies advancing or progressing, and the subsequent **Old French** *proces*, which implies continuation or development. “Process” therefore means following a sequenced method to accomplish a result. Accordingly, a software process is comprised of a series of activities to design and develop software. The focus is on the construction process (to design and develop software artefacts and systems), rather than the output (the created software artefact or system). A software process model is an abstraction of such a software process. Software is developed using software processes, which are **subsequently** based on software process models. The development of software and software systems is planned and executed according to a software life cycle model, which is an abstraction of the **software life cycle**. The software life cycle includes all the steps and activities of a software project. It spans from initialization of the software to its withdrawal, i.e., from inception to maturity and beyond, and consists of all the phases through which a single software artefact or an integrated software-intensive system develops. Software processes and life cycle models aim to facilitate coordination and management of the various and complex activities that are involved in the development of software (Kneuper, 2018).

Software life cycle
The software life cycle describes the steps of a software project, from initialization to withdrawal.

Basic Software Processes and Life Cycle Models

The various software processes and life cycle models that manage software development projects and are applied in software engineering include, e.g., waterfall models, the V-model, component or matrix-based models, iterative, incremental, and evolutionary development models, and agile and lean development. Regardless of the specific model applied, a life cycle model typically includes a sequential, iterative, or evolutionary arrangement of the following generic phases:

- a feasibility study
- requirements elicitation
- an investigation of the existing software and associated infrastructure
- analysis and object design
- design, including the software and system design
- implementation of the software and system
- verification and maintenance

The feasibility study involves investigating the requirements that are not being met by the current software in use (if applicable), and the requirements that need to be met by the new software. Requirements elicitation entails collecting and analyzing business requirements and translating them into functional and technical requirements. The investigation of the existing software and associated infrastructure entails detailed scrutiny of the flaws of the current software and infrastructure (if applicable). It also aims to identify additional functional and technical requirements, any possible constraints, data types, volumes, etc., of the software to be developed. During analysis and design, the aim is to understand the improvement that the new software should bring about and the design that can best achieve it. Factors, such as relevant objects, input and output, processes converting input to output, security, privacy, backup provisions to be made, the definition of testing, and implementation plans, are also considered. Implementation refers to the actual implementation, testing, and use of the new software in the business environment; it includes change over from the old to the new (if applicable), drafting documentation, and conducting end-user training according to defined privacy and security protocols. Finally, verification and maintenance consist of evaluation and assurance of continued optimal use of the software by the users until withdrawal.

The Management of IT Projects in the Software Engineering Discipline

The software engineering discipline comprises the specification, design, development, management, and evolution of software and software-intensive systems. IT, including software systems, is applied to solve problems in various and diverse industries, e.g., manufacturing, education, logistics, production, government, finance, health care, and analytics. The intrinsic high complexity of software systems necessitates suitable application of engineering principles; accordingly, software engineers apply relevant methods and techniques from engineering fields to solve problems efficaciously with software (Sommerville, 2011). Software engineering projects are regarded as a type of an IT project; they follow a project management approach and apply IT project management principles to develop artefacts and systems.

Project management is the identification and organization of milestones, tasks and activities, timelines for completion, allocation to responsible and accountable individuals, and monitoring of work performed in comparison to plans and schedules. Similarly, IT project management is progressing towards materializing an IT (e.g., software) artefact; it defines the phases to follow in order to visualize, design, develop, implement, and maintain software. It also entails the “procedures, techniques, tools, and documentation” used to “plan, manage, control, and evaluate” IT projects and the associated software artefacts and systems they produce (Avison & Fitzgerald, 2006, p. 24).

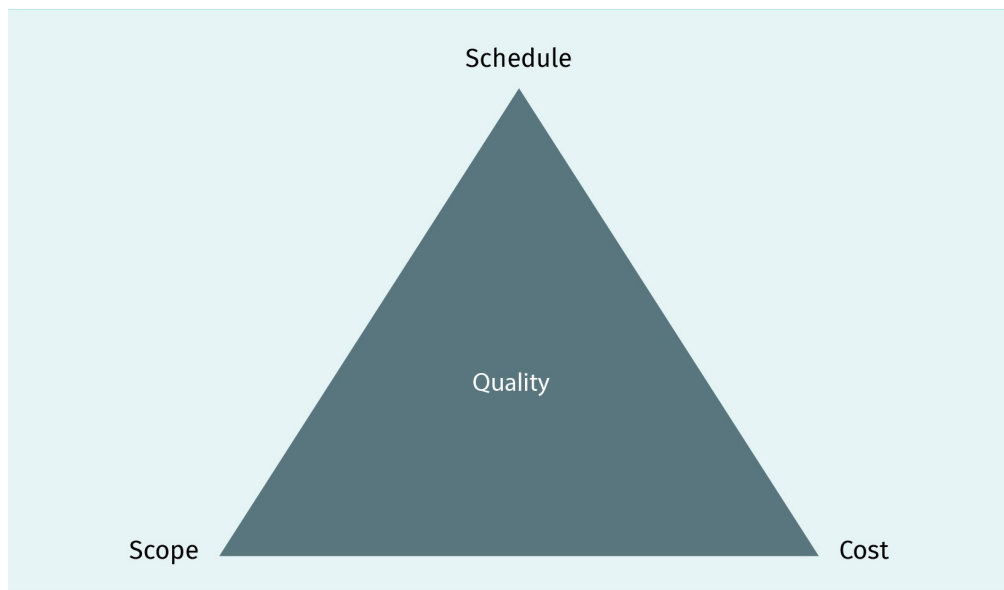
According to Münch et al. (2012), processes and models that are suitable and appropriately applied aim to reduce the complexity of large software development projects, ensure that all members of a development project work together in a coordinated manner, and facilitate the development of software that is consistent with quality criteria, as well as within time and budget constraints. Successfully executing such projects requires the rigorous application of a suitable approach and appropriately managing the associated risks and challenges. However, despite the rigorous planning and execution of IT projects, they often fail. Challenges and issues that are cited as causes of IT project failure include

- unclear and changing requirements (Hussain & Mkpojiogu, 2016).
- the requisite to meet the needs of various and diverse stakeholders (Alreemy et al., 2016).
- a multitude of diverse (even disparate) internal and external requirements and stakeholders that influence the design and behavior (de Oliveira & Rabechini Jr, 2019).
- the socio-technical nature caused by the interconnectedness of technological platforms, such as hardware and software, as well as human users (Teubner, 2019).

It is unfortunate that IT project failures, especially in the software domain, are often rationalized or ignored, resulting in the repetition of mistakes by others or even by the organization involved and affected.

To improve the rate of (or, at the least, achieve) success, we must first define it. In project management guides and standards, such as the PMBOK Guide (Project Management Institute, 2013) and PRINCE2 (Axinte et al., 2017), project success is generally measured by the project triangle. This triangle refers to three constraints that must be balanced when planning and executing a project, i.e., the schedule (timeline of activities), the scope (the boundaries of what is included versus what is to be excluded), and the cost (the allocated budget for the project). Balancing these aspects means that the extension of one of these elements will inevitably impact the others. For example, an extension in the schedule will cost more, or a scope increase will inflate both the schedule and the cost. More recently, a fourth element has been added: quality. Quality can be put in the middle of the triangle, indicating that quality is dependent upon schedule, scope, and cost, and vice versa.

Figure 1: The Project Triangle—Indicating Project Constraints



Source: Venter, 2021.

Taking the aforementioned challenges and issues into account, it is clear that IT project success can potentially be very difficult to concisely determine and define. For example, when a project serves a supporting role to the core organizational functions, added factors, such as business, political, cultural, and social undercurrents, must be considered in addition to the technical complexity of the project. This is typical for software projects and, as a result, the boundaries and scope definitions of software projects tend to be fluid, sometimes to the detriment of the project. Also, constant changes and rapid technological advancements, as well as the need to be up-to-date with the latest technological developments, often lead to rapidly changing expectations and requirements. Additionally, issues, such as the need to accommodate both the old and the new simultaneously and having to replace legacy systems without interrupting core business functions, pose more real threats of project failure. Project risks refer to uncertainties in conditions that can influence a project outcome positively or negatively. Risks are managed using

- risk identification.
- risk analysis in terms of probability.
- size.
- potential risk mitigation, risk avoidance, or risk transference actions.
- implementation of defined mitigation, avoidance, and transference actions (as needed).
- monitoring to determine if these were sufficient, or whether they should be adjusted.

Specific and unique sources of risks for IT projects are technological, organizational, and user-dependent (Taherdoost & Keshavarzsaleh, 2015). Technological risks include, e.g., the technology of the product or service failing to integrate or interface sufficiently with existing systems or platforms as intended, underestimation of the number of users, and lacking a system capable of scaling up with demand. Organizational risks include, e.g., lack of strategy for technology acquisition, resulting in insufficient resourcing for the

project or underestimation of staffing support requirements. Often, IT departments must deliver projects reactively, with little room to influence user requirements, as these projects are being generated by leadership or business units with little understanding of the time and effort required to build an IT solution.

Project Management in Software Engineering Processes

In contrast to industrial production processes where activities can be planned upfront in terms of, for example, duration and resource requirements, planning for software projects tends to be unpredictable and remain fluid throughout the process. In particular, the exact costs, a concrete functional scope, and technical design of a software system are often only known in retrospect. Furthermore, major contributing factors of project risk in software projects include the interconnectedness, complexity, and intangibility of software systems.

Even though software development is mostly a knowledge-driven, user-centric, and social process, software projects are most often planned and launched according to an overarching technical objective (e.g., the introduction of a self-care portal for customers of an insurance company), meaning that it is nearly impossible to specify all the requisite functional requirements and integration points accurately from the onset. The conceptual nature of software makes it difficult to review the progress and quality of both the individual elements and the integrated components during development and prior to use. New requirements therefore tend to surface during development, even after initial use; the requirements relevant for users and customers typically only become known after stakeholders have seen a first version of the system. Diverse stakeholder groups may also have diverse or conflicting requirements that may only emerge during development and after initial use. In addition to numerous stakeholders, factors, such as changing legal requirements, technological developments, and unpredictable markets, can significantly influence the requirements for software systems during a software project.

Software development's success continues to be driven by the degree to which a developer understands the business requirements of end-users (Green et al., 2010; Leffingwell, 1997; Sawyer et al., 1997). Common causes of failure include the inappropriate specification and management of a customer's requirements, inconsistent or incomplete requirements, expensive late changes, and misunderstandings between the involved and affected stakeholders. Sommerville (2001) suggests that business requirements should be verified in terms of validity, consistency, completeness, realism, and verifiability. These terms are defined as follows:

- Validity refers to the inclusion of appropriate and relevant functions, where user communities may have to compromise if these are too diverse.
- Consistency refers to requirements that do not conflict or have contradictory constraints or descriptions.
- Completeness ensures that all functions and constraints that the user intended are clearly defined.
- Realism means that requirements can be implemented within budget, time, and technological constraints.

- Verifiability refers to the ability to use assessment criteria to confirm whether the software will meet specified requirements after implementation.

Development teams must have both a business and a technical understanding of the requirements formulated by stakeholders in order to deliver a highly usable and purposeful software system. As software systems are generally integrated into complex application landscapes via a multitude of technical interfaces, development teams must be able to identify both business and technical relationships across organizational and system boundaries. Thus, project teams with business understanding, as well as technical knowledge of applications, can deal more effectively with the intricate combination of interrelated and interdependent business, functional, and technical requirements.

When software projects focus largely on the technology without addressing the benefits for the end-user, it is referred to as “technology-centeredness”. Similar problems occur when challenges to technology are only superficially explored, thereby leading to a high risk of project failure. It is more convenient (and exciting) for developers to discuss these interesting topics and put new technologies into use than to confront the business and technical problems of users. In some cases, this leads to systems being delivered that reflect the latest technology trends but are not designed according to the actual needs of the users.

Finally, the lack of communication and coordination between those involved in and affected by a software system also causes project failures. Development projects for software systems can involve various (diverse) departments (e.g., marketing, sales, IT application development, external consultants, and the legal department), and each organizational unit might have unique ideas and objectives that allow them to accomplish their tasks more effectively.

1.2 A Historical Overview

Boehm (2006) argues that software engineering differs considerably from other types of engineering. Engineering in its basic form tends to be a relatively static field, e.g., basic electronics or chemicals do not change their basic structures over time. However, the software elements that we engineer continuously change and evolve over time. From a historical viewpoint, and as a result of these evolutions, software process models and life cycle models are positioned in, and can be explained from, the different perspectives of the prevailing **paradigms**, i.e., evolved worldviews that resulted in new models as well as existing models that were adapted accordingly.

Paradigm

A paradigm refers to an accepted model or pattern and represents the way people perceive, view, and explore their world.

Thomas S. Kuhn (1962) introduced the concept of paradigms and paradigm shifts in his book *The Structure of Scientific Revolutions*. Paradigms are still applied today as a lens through which to explore and explain phenomena, such as the processes and models that are applied to plan and structure development tasks and projects. Therefore, even though the basic elements of software processes and life cycle models are **fairly alike**, the fundamental paradigm of the respective software process and life cycle model guides its choice and determines how it is to be implemented and executed.

Paradigm shifts occurred over the last few decades in the way that software was (and is) perceived and used. Shifts also occurred in the way that software is designed, developed, implemented, and maintained, and software processes and life cycle models evolved accordingly. The emergence of a new paradigm does not necessarily render the previous one invalid; Kuhn (1962) argued that it merely reflects new and different ways of understanding and doing. Therefore, various paradigms can co-exist, and one will be chosen over another based on, for example, organizational culture, specific requirements, or particular circumstances.

Similarly, the disciplines of software development and software engineering evolved from purely algorithmic and one-off activities. Over time, they became more formal and structured, and matured into an acknowledged discipline, rooted in both the engineering and the business world alike, over the past 60 years. The different processes and life cycle models that are currently being applied in the field reflect the various viewpoints used to understand software requirements, as well as to design, develop, and maintain software. As an example, Iivari et al. (2000) suggest that development approaches are rooted in either a functionalist or a non-functionalist paradigm. They positioned structured models, e.g., waterfall models, and also typical iterative, incremental, and evolutionary models in the functionalist paradigm, arguing that these focused mostly on technological aspects of the software, to the detriment of the end-user. More recently, Boehm (2006) also argued that software engineering has evolved to the extent that it acknowledges that software can only be “useful to people” when it is engineered using relevant sciences, such as “the behavioral sciences”, “management sciences”, and “economics”, as well as “computer science” (p. 13).

Programmable Computers: The Origin of Software

General-purpose programmable computers and accompanying software originated, in theory, in 1834. Charles Babbage and Ada Lovelace devised them conceptually when they considered the use of mechanical machines for complex technical calculations and designed the “Analytical Engine” (Babbage, 1864, p. 186). If they had been able to build it at the time (lack of resources and technological advancements prevented them), it would have been the first general-purpose programmable computer (Wilkes, 1992). An informal program (algorithm) for the analytical engine was written by Lovelace in the form of commentary added to a description of the analytical engine in 1842 (Menabrea, 1843).

About a century later, sufficient resources became readily available and technological advancements were at such a level that the first programmable computers could actually be built. These were for government and military use at first, but they were also adapted for civil use after the Second World War (Zuse, 1980). During the Second World War, different countries (i.e., Germany, Great Britain, and the United States) were concurrently busy studying computer technology. Without knowing of each other’s work, their respective designs were quite similar, as they were mostly inspired by the work of Babbage and Lovelace (Aiken & Hopper, 1946; Eckert et al., 1951; Flowers, 1983; Zuse, 1980).

After the war, computer technology advanced astonishingly quickly, from the “51 feet long and 8 feet high” (Aiken & Hopper, 1946, p. 386) Mark I computer intended solely for government use, to inexpensive and relatively small (for the time) single-chip microprocessor

architecture. This ultimately resulted in portable and personal computers that are, nowadays, relatively cheap and therefore widely used in various institutions, including government and military organizations, businesses, universities, schools, and private households. Standardized software processes and life cycle models evolved accordingly, albeit somewhat slower by comparison.

The Evolution of Programming for Software

Since programmable hardware was being built beginning in the 1940s, supporting software became necessary soon thereafter. For this, initial programming languages, e.g., Konrad Zuse's *Plankalkül* language, were largely algorithmic and based on Boolean logic (Zuse, 1980; Boole, 1847). Wallace John Eckert published a pattern language that was essentially viewed as the first programming methodology in his book *Punched Card Methods in Scientific Computing* in 1940 (Eckert, 1940). Commercialization and the wider use of computers soon led to acknowledgement that standardization was required in the programming field. Thus, compiler programs were born, such as the one completed by Grace Hopper in the early 1950s, which is considered to be the first compiler program (Hopper & Mauchly, 1997).

John Pinkerton, an engineer at the Lyons Electronic Office (LEO), also realized that low-level and repetitive programming tasks could be bundled into a library of common, reusable routines. Before long, programmers Grace Hopper, Robert Bemmer, and Jean Sammet, influenced by the work of John Backus, created the powerful, business-oriented programming language Cobol, and the predecessors to open-source software (the SHARE organization) and the idea to establish and outsource software development as a business emerged. This opportunity was seized by British programmer Dina St Johnson when she founded the first software development house (Booch, 2018).

The Rise of Structured Software Engineering

It is evident from the preceding discussion that software engineering developed over many years, starting with typical craft-based, trial-and-error approaches. These make-and-fix approaches continued to produce expensive artefacts, quite often behind schedule. Development approaches evolved over time into methodological (engineering-based) approaches that are more time-based, resource efficient, and structured.

Benington (1983) was the first to present a formal and structured description of sequenced activities whereby to develop software; it was presented at a 1956 symposium dedicated to advanced programming methods for digital computers. However, it was only in the late 1960s that the term “software engineering” was formally introduced. The term is said to have been coined by Friedrich Bauer in 1968 at the first NATO software engineering conference, where the organizers of this conference recognized software as becoming an integral part of communities and organizations alike, and structured software development approaches became more prominent (Naur & Randell, 1969). Even so, the term “software engineering” may have emerged earlier; a letter authored by Anthony Oettinger was published in 1966 in the *Communications of the ACM*, which used the term “software engineering” to distinguish between computer science and the practice of building of software (Oettinger, 1966). Earlier still, an advertisement was published in June 1965 in an issue of

Computers and Automation, seeking a “systems software engineer”. However, unpublished oral history states that Margaret Hamilton actually coined the term “software engineering” in the early 1960s to distinguish her work from that done in the hardware engineering field (Booch, 2018). From the 1960s to the 1980s, the field of software development and engineering advanced rapidly. The following advancements occurred in this time period (Booch, 2018):

- The concept of modular programming was born.
- Structured programming was conceptualized.
- The programming language Pascal was invented to support structured (functional or procedural) programming.
- An object-oriented language (Simula) was invented.
- Various ideas related to information hiding, abstract data types, entity-relationship modeling, software engineering methodologies, and structured analysis and design emerged.
- A variety of programming languages materialized, and computers and software became more available and accessible.
- Businesses started to use computers, software, and programming languages to improve and optimize various organizational aspects.
- Software-intensive and distributed systems appeared to replace stand-alone and independent software and software artefacts.
- Wide-spread problems related to quality, privacy, and security emerged due to the speed at which these industries were growing.
- Structure and formalization was needed to control and manage the growth, and various models were progressively introduced.

The Evolution of Software Process and Life Cycles

The software development life cycle (SDLC) model, as introduced by Royce (1970), is still viewed by many as the first and traditional software development approach as it greatly influenced software development practices (Avison & Fitzgerald, 2006). The SDLC model was, however, more generally termed, similar to the set of sequential development activities presented by Benington in 1956 (Benington, 1983). It comprises of activities, such as the definition of system and software requirements, design and analysis, programming and coding, testing, and continued operations. Royce (1970) argued that these successive development phases should ideally be iterated before proceeding to the next phases, rather than executing them strictly sequentially. The term “waterfall”, which is still widely associated with the traditional SDLC model, was introduced by Bell and Thayer (1976) when they referred to it as a top-down approach and suggested that sequences of activities develop software.

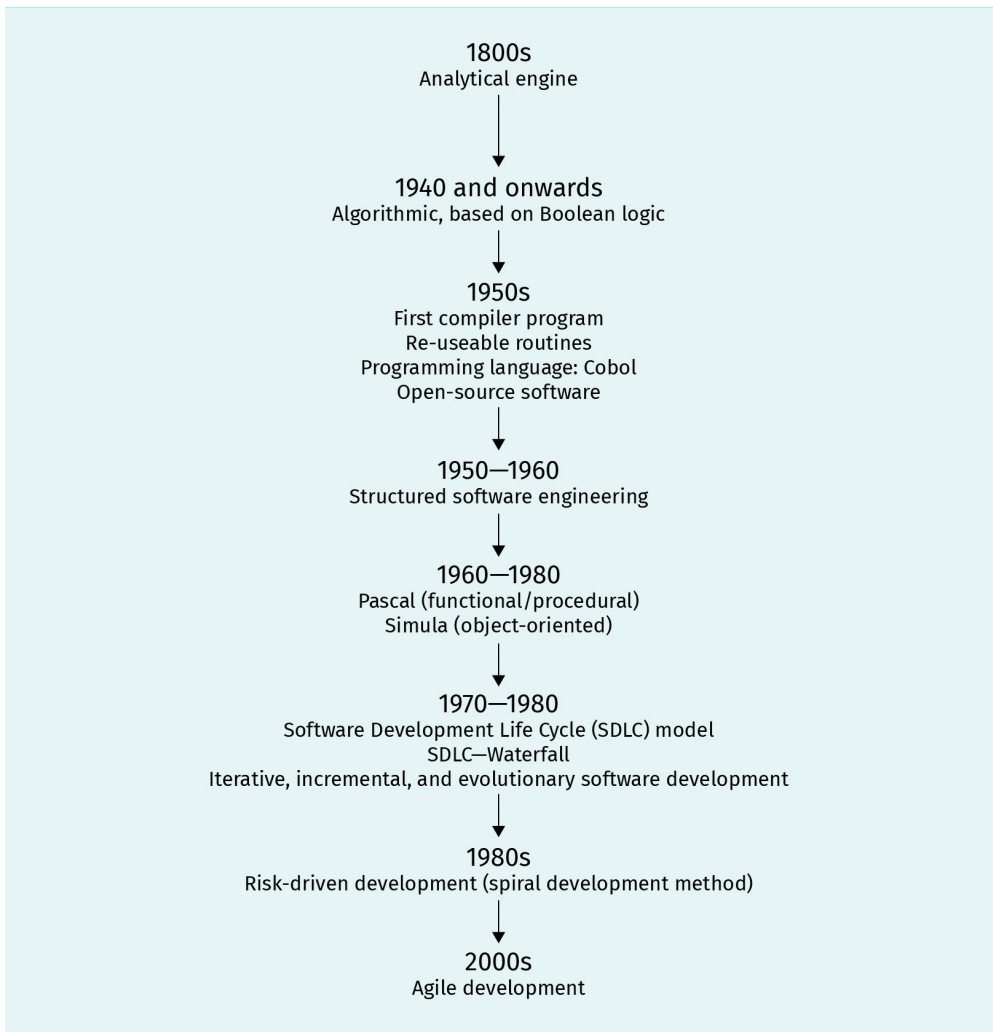
According to Edmund (2010), general iterative, incremental, and evolutionary software developments were originally inspired by the work of Shewhart, who suggested in the 1930s that problems are best solved by following a plan-do-check-act (PDCA) cycle, which later became known as the four basic quality improvement steps. Randell and Zurcher (1968) say that they recommended and presented evolutionary software development in the late 1960s at a congress, stating that such an approach may result in improved software products. They argued that, since issues and problems can only truly be detected

once a product has been built and is being used, issues can be discovered and resolved prior to continuing to the more intricate and complex levels of design and development when software is designed, developed, and evaluated iteratively and incrementally.

Boehm introduced risk-driven development, in the form of the spiral development method, in the late 1980s. This method incorporates any development methodology, or a combination thereof, and consists of an indeterminate number of iterative loops that set objectives, assess and reduce risk, develop and validate, and plan for the next phase (i.e., loop), until completion of a successful artefact (Boehm, 1988). This method is risk-driven, rather than document or code-driven.

While Agile development and the principles behind it were formally introduced in the early 2000s (*The Agile Manifesto* was published in 2001), the Scrum method was presented earlier. “Scrum” was introduced by Takeuchi and Nonaka in 1986. Soon after, Ken Beck introduced Extreme Programming, and Johnson and Fowler developed refactoring (Booch, 2018). Agile development consists of techniques that are referred to by the founders of the Agile development principles as “light” and support rapid development (Highsmith, 2001). It successfully facilitates iterative and incremental development, as well as continuous verification of requirements and subsequent evolution of products (Hijazi et al., 2012). Agile, in the broadest sense, refers to a type of cultural approach whereby to rapidly design and develop user-centric software. A high-level historical overview is shown below.

Figure 2: High Level Historical Overview



Source: Venter, 2021.



SUMMARY

Software engineering is the engineering of software for organizations and individuals. Different types of engineering are applied, but they all follow a pre-defined software process and life cycle model. Software engineering trends have been emerging over the years, and the field still continues to evolve. Programming practices evolved from algorithmic, code-and-fix approaches, to formalized and structured development methods, and software engineering now comprises of a multitude of processes and models. Software processes and life cycle models consist of sequential, iterative, and evolutionary arrangements of a number of generic phases. These involve a feasibility study, requirements elicit-

tion, investigation into existing software and infrastructure, analysis and object design, software and system design, software and system implementation, and verification and maintenance.

Project management and engineering principles are applied to design, develop, and maintain software, from initialization to withdrawal. They reduce complexity and risk and facilitates communication among team members. Still, failure of software (and IT) projects remains high. Causes of failure continue to be identified, and management of these projects continues to be researched and refined.

Software engineering, as a discipline, emerged in the 1960s; however, computer hardware has been used since the early 1900s, and programming practices advanced as hardware (and application of technology) advanced. Software processes and life cycle models can be explored and explained from the different perspectives of prevailing paradigms. Similarly, evolutions that occurred in the field over the past decades give insight into the way software was (and is) perceived and used.

UNIT 2

SOFTWARE PROCESS DEFINITION AND MODELING

STUDY GOALS

On completion of this unit, you will have learned ...

- about notations and the role of meta-models.
- how to apply the Unified Modeling Language (UML) to model software-intensive systems.
- how the Systems Modeling Language (SysML) differs from UML.
- about the relevant notations to model interactions between business processes.
- to use the detailed level notation Business Process Model and Notation (BPMN).

2. SOFTWARE PROCESS DEFINITION AND MODELING

Introduction

Software processes are dependent upon internal and external factors, people, and circumstances; they are complex, unpredictable, and challenging to rationally describe. Activities, resources, and constraints associated with software processes are difficult to manage (Bendraou et al., 2010). Software process definition and modeling enables optimal design, development, and implementation of robust software; it also facilitates optimal management of risk as well as verification and maintenance of implemented software. Models help to understand, visualize, and communicate desired structure, behavior, and architecture of software-intensive systems, and they also guide construction and documentation of decisions (Booch et al., 2005).

Software modeling can be approached from either an algorithmic or an object-oriented perspective. In the algorithmic view, procedures and functions form the building blocks of the software; in the object-oriented view, objects or classes form the main building blocks of the software. Object-oriented development is applied widely to develop modern-day software. In this, the following modeling components are used: classes, objects, attributes, and operations. Identified classes and objects, their associated attributes and operations, and how they relate to one another to conceptually describe a problem and solution are noted in a standardized modeling language in order to document the results of the object-oriented analysis (OOA) and object-oriented development (OOD). The Unified Modeling Language (UML) and Systems Modeling Language (SysML) are standardized object-oriented notations that are widely applied. Various other notations to model interfaces between business processes, as well as the business processes themselves, are relevant in software engineering. These are necessary to visualize the interconnectedness of the software systems, and optimally manage the risks around it. In addition to formal notations, the “napkin” notation is often used to draw diagrams that are easy to read and understand from the perspective of non-technical audiences, e.g., to present a simplified, high-level, or non-formal diagram to management. The most widely used notations are explained in this unit.

2.1 Modeling Notations and Meta-Models

The Unified Modeling Language (UML) is a graphical modeling language that was developed in the early 1990s with the emergence of object-oriented development practices. It is a software-centric modeling language that facilitates modeling from an object-oriented viewpoint. According to Bendraou et al. (2010) UML provides “a rich set of notations, diagrams, and extension mechanisms” (p. 662). It has both advantages and drawbacks, but it remains “undeniably one of the most adopted modeling of this decade” (p. 662). It is widely applied to model software-intensive systems.

The SysML is also a universally accepted modeling standard. It is systems-centric rather than software-centric, and is applied to model systems as well as system-of-systems. Friedenthal et al. (2012) refer to SysML as “a general-purpose graphical modeling language for representing systems that may include combinations of hardware, software, data, people, facilities, and natural objects” (p. 3). It was derived from and extends a portion of UML.

Software systems, as part of a bigger organizational landscape, are also required to be modeled in a similarly cohesive manner; hence, interfaces and integration with, for example, related and interconnected business and systems, must be analyzed and illustrated. For this, high-level modeling notations that cover multiple processes, as well as individual detailed level notations, are applied. Prominent notations in this regard are those that include value chains and process landscapes, e.g., the Multi-View Process Modeling Language (MVP-L) and the Business Process Model and Notation (BPMN). All notations are defined by applicable meta-models.

The Role of Meta-Models

A meta-model (or surrogate model) is an abstraction of a model. A meta-model, as an abstract representation, defines the notations to be used to model a process. Meta-modeling entails the analysis, construction, and development of elements, such as the frames, rules, constraints, and theories; these should then be explicitly applied when modeling specific problems and solutions. A meta-model describes and illuminates the properties of the model; a model always conforms to a unique meta-model. Kneuper (2018) states that meta-models are distinguishable through their unique properties. They differ in terms of the following:

- levels of abstraction and detail
- degrees of structure
- degrees of formality
- the means to support process enactment, execution, and simulation
- use of either graphical or text-based notation

Kneuper (2018) differentiates between process interaction notation, notation for process-internal structure, and combination notation. He classifies life cycle diagrams as process interaction notations, and notations that are typically applied to model requirements and business processes are classified under process-internal structures. The Object Management Group (OMG) (n.d.) developed a meta-modeling architecture that supports the description of widely used languages, such as UML and BPMN, referred to as the Meta-Object Facility (MOF). MOF is essentially a meta-meta-model, as it describes the notation used for meta-models (Kneuper, 2018). MOF supports the Software Process Engineering Metamodel (SPEM) and provides a basis for tool support. SPEM was introduced by the OMG and embodies a process engineering meta-model and a conceptual framework to provide necessary concepts to model, document, present, manage, interchange, and enact development methods and processes (Münch et al., 2012).

MOF is published as the international standard ISO/IEC 19508:2014 (International Organization for Standardization, 2014). It is a Domain Specific Language (DSL) that provides a type system for use in the Common Object Request Broker Architecture (CORBA) architec-

ture. CORBA facilitates communication of systems deployed on diverse platforms. MOF is typically used as a four-layered architecture, i.e., M3 to M0. It provides language (on M3) used to build meta-models (referred to as M2 models), which then describe elements of the M1-layer (M1 models). M0 is the data layer, which is used to describe objects as they are in the real world. MOF meta-models are typically modeled as UML class diagrams. The M3 layer can be referred to as the meta-meta-model. The M2 layer is made of the meta-model (e.g., UML, BPMN, and SPEM). The M1 level is then the instance of the meta-model, i.e., the model itself, while M0 level is the instance of the model.

Unified Modeling Language

Unified Modeling Language (UML) is not a programming language; it is a graphical language, or notation, used to model software-intensive systems. It helps to document blueprints for software systems as it can visualize, specify, construct, and document the systems, by focusing on the conceptual and physical representation thereof. It applies a set of rules and semantics to specify and communicate the structure and logic of software-intensive systems (Avison & Fitzgerald, 2006). UML was devised by Grady Booch, Ivar Jacobson, and James Rumbaugh between 1994 and 1996, to facilitate modeling from an object-oriented perspective; it is therefore used to model analysis and design concepts of software systems in an object-oriented fashion. They formulated UML in an attempt to standardize notations in order to remove confusion related to varied notations applied, and increase adoption of object-oriented techniques (Booch et al., 2005). UML was initially developed by integrating the pre-existing Booch method (of Grady Booch), the object modeling technique (of James Rumbaugh), and object-oriented software engineering (of Ivar Jacobsen), with elements of other applicable methods and the assistance of large corporations such as IBM, Unisys, Oracle, Microsoft, Hewlett-Packard, and Digital (Baumann et al., 2005). The Object Management Group (OMG) adopted UML in 1997 and it is now a vendor-neutral and evolving public standard. The latest versions of UML (the most current version is UML 2.5) also support the modeling of business processes for the context of software engineering. When considering UML in its entirety, it is almost incomprehensibly large. In reality, only a portion of it is typically applied. The diagrams that are most often used in process modeling are typically class diagrams, state machine diagrams, and activity diagrams (Bendraou et al., 2010).

Core components of UML

Firstly, in object-oriented modeling (and UML), it is important to understand the difference and the relationship between an **object** and a **class**. An object is a concrete manifestation of a class or an actual instance of a class. A class describes a set of common objects. Each object has a unique identity that distinguishes it from other objects, a state referring to data (i.e., a unique set of value entries of its attributes) associated with it, and behavior referring to actions (i.e., operations) that it can perform, or that can be imposed upon it by other objects. Secondly, in order to apply UML effectively, one must understand the following three elements of UML: its basic building blocks, the rules dictating how they may be put together, and the common mechanisms that are applicable throughout. Booch et al. (1999) state that the basic building blocks of UML are as follows:

Object

An object is something concrete and representative of the problem or solution space.

Class

A class is an abstract representation of an object and describes a set of common objects.

- structural, behavioral, grouping, and annotational things, which are essentially abstractions of logical or physical entities that are representative of the problem or solution being modeled
- relationships that tie all the elements together
- diagrams, which are groupings of meaningful collections

Then, the rules must be applied to ensure harmonious models; they are semantic rules to describe and define names, scope, visibility, integrity, and execution. Furthermore, common mechanisms should be applied in terms of specifications, adornments, common divisions, and extensibility mechanisms. All of these are applied to draw diagrams that represent views of the problem space and a proposed solution.

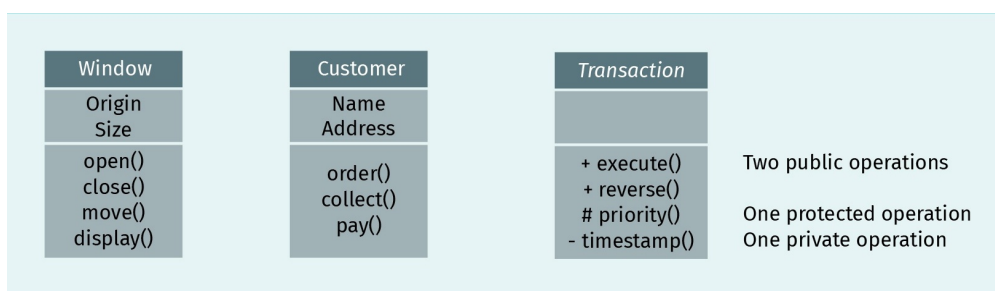
Structural things, or abstractions, represent conceptual (logical) or physical elements. The basic structural elements to include in a UML model are made up of conceptual and physical elements. The conceptual elements are classes, collaborations, use cases, active classes, and interfaces. The physical elements are components and nodes. These are also broadly referred to in UML as classifiers. A classifier is a meta-class describing a set of instances sharing common features, whereas features declare instances of classifiers' structural or behavioral characteristics.

The structural, conceptual, and physical things in UML

A class describes a set of objects that share similar attributes, operations, relationships, and semantics; it implements one or more interfaces. A class is drawn as a rectangle separated into three distinct parts, typically including its name, attributes, and operations.

The figure below shows examples of three different classes.

Figure 3: Examples of Classes

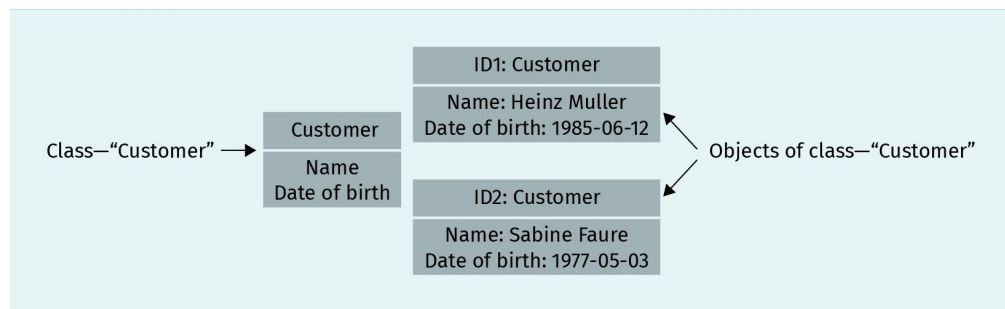


Source: Venter, 2021.

The first class (1), which has the name “Window”, indicates that a “Window” has the attributes “origin” and “size” and can perform the operations “open”, “close”, “move”, and “display”. The second class (2) has the name “Customer” to indicate that its objects will contain information about customers; it specifies that a “Customer” has the attributes “name” and “address”, and it indicates that a “Customer” can perform the operations of “ordering”, “collecting”, and “paying”. In the example above, the third class (3) shows how notations can also be used to indicate the status of the operations. For example, in the

class “Transaction”, the public (+), protected (#), and private (-) operations are specified. The name of the class “Transaction” is written in italics to indicate that this is an abstract class. These examples are simplified for the purpose of illustrating how to model them, as classes typically contain more attributes. Objects are a special form of a class; they represent concrete manifestations of classes. When objects are modeled, the attributes are associated with actual values. An example is shown below.

Figure 4: Example of a Class and Objects of the Class



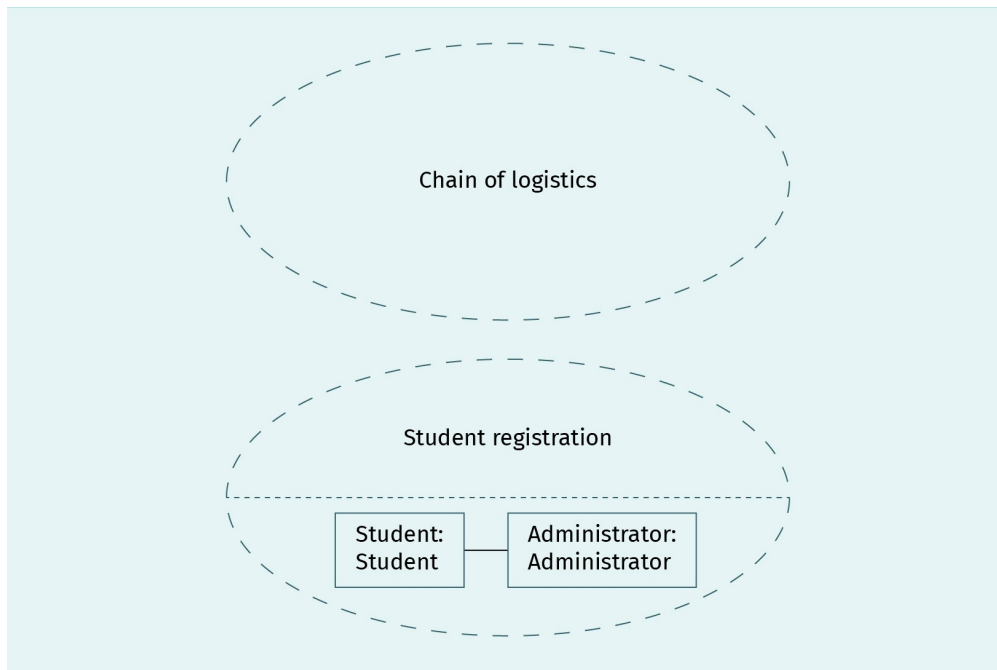
Source: Venter, 2021.

A collaboration defines an interaction; it is a combination of roles and other elements, working together cooperatively. Collaborations are bigger than the sum of the elements, so they have structural dimensions and behavioral dimensions; this represents the implementation of patterns making up a system. It is drawn as an ellipse with dashed lines and typically includes its name. It can also be indicated by both its name and relevant roles and connectors. Details are suppressed. The collaboration “Chain of logistics” is shown below. In this example, the “Chain of logistics” refers to a **business process** describing the chain of logistics of a company. Also, the collaboration “Student registration” refers to the process where a student and administrator collaborate to register a student.

Business process

A sequence of activities aimed at reaching an organizational goal.

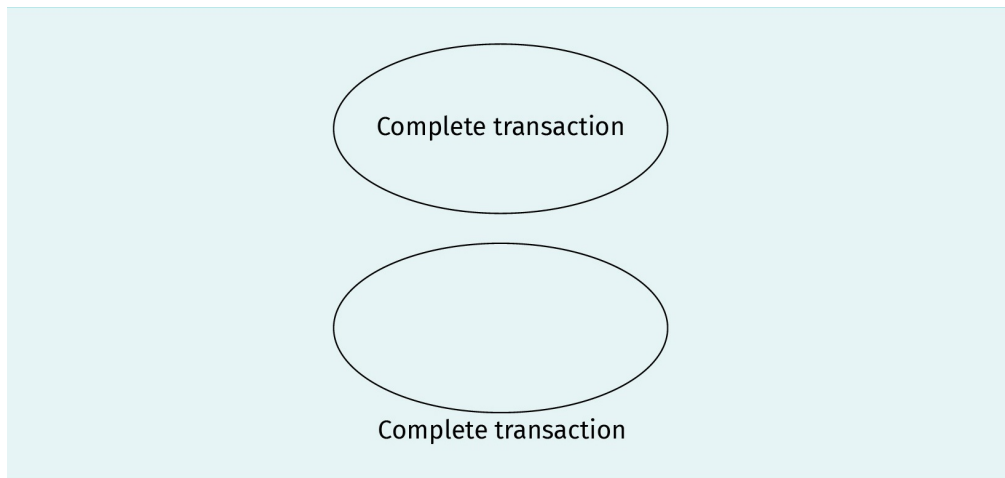
Figure 5: Examples of Collaborations



Source: Venter, 2021.

A use case describes a set of sequenced actions performed by a system that yield a tangible, valuable result for a specific actor. It is applied to structure behavioral things and is realized by a collaboration. It is drawn as an ellipse with solid lines. It typically includes its name in the ellipse; however, the name can also be shown underneath the ellipse. A use case “Complete transaction” is indicated below. In this example, the use case “Complete transaction” refers to the collective actions that must be performed to complete a specific transaction as it is encapsulated in and required for a specific business process and for an actor. Both notations are shown—the first includes the name in the ellipse, the second shows the name below the ellipse.

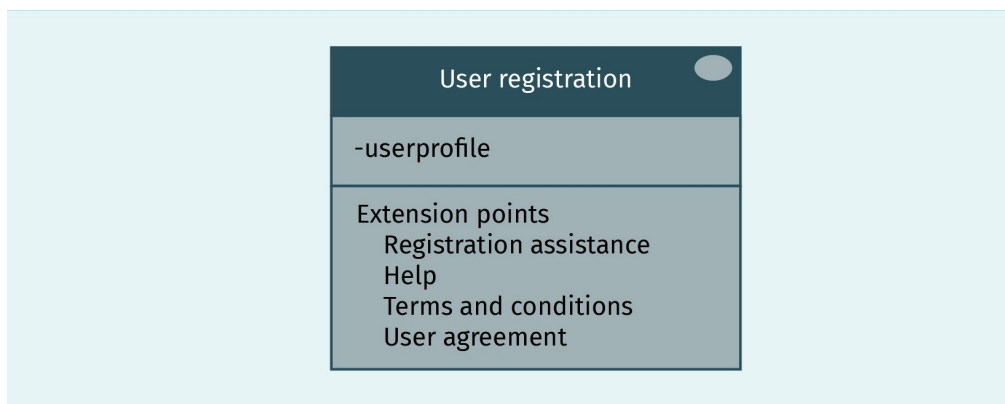
Figure 6: Examples of Use Cases



Source: Venter, 2021.

A use case can also be indicated using a standard rectangle with an ellipse icon drawn in the upper right-hand corner of the rectangle. Furthermore, it may also include separate compartments to show additional features. In the example below, a use case “User registration” includes additional features to register a user profile.

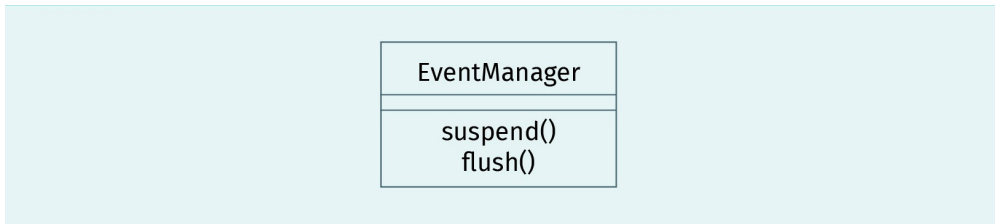
Figure 7: Example of a Use Case



Source: Venter, 2021.

An active class is a class with objects that owns one or more processes or threads; it can initiate control activity and its behavior is concurrent with that of other elements. It is drawn as a class, but with a heavy outline, and may include its name, attributes, and operations. The active class “EventManager” is shown below.

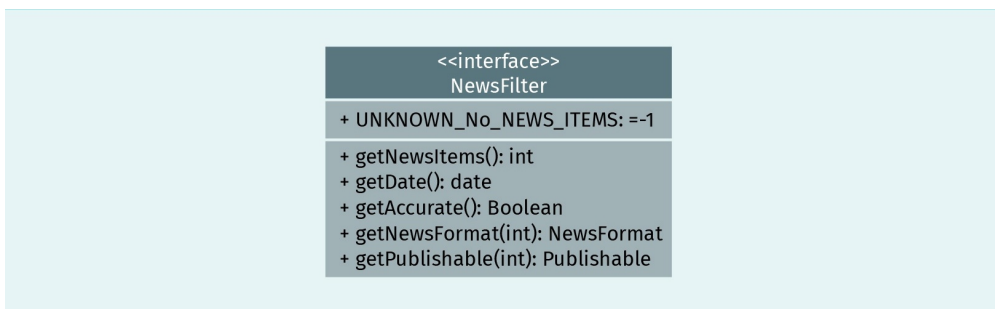
Figure 8: Example of an Active Class



Source: Venter, 2021.

An interface declares features and obligations. It constitutes operations specifying a service that realizes a classifier, or that is required by a classifier; it is a declaration of the externally visible behavior. It may represent complete or partial behavior and defines operation specifications (signatures), but not operation implementations, and can also indicate constraints or protocol specifications. It is drawn as rectangle with the keyword <<interface>> and its name; it may also include associated features and obligations, as is illustrated below.

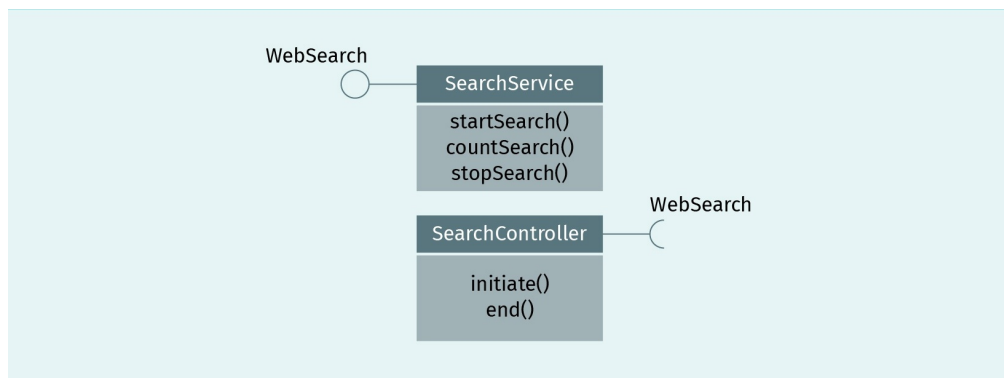
Figure 9: Example of an Interface



Source: Venter, 2021.

An interface's operations are either realized by a classifier, or required by a classifier that is dependent upon it. Realization is drawn as a circle connected to a line, i.e., a "lollipop". Dependency is drawn as a solid line attached to a half-circle or socket. The example below shows that the classifier "SearchService" is realized by the interface "WebSearch"; "SearchService" manifests because of the interface "WebSearch". It also shows that the classifier "SearchController" requires the interface "WebSearch". The examples below show interface usage.

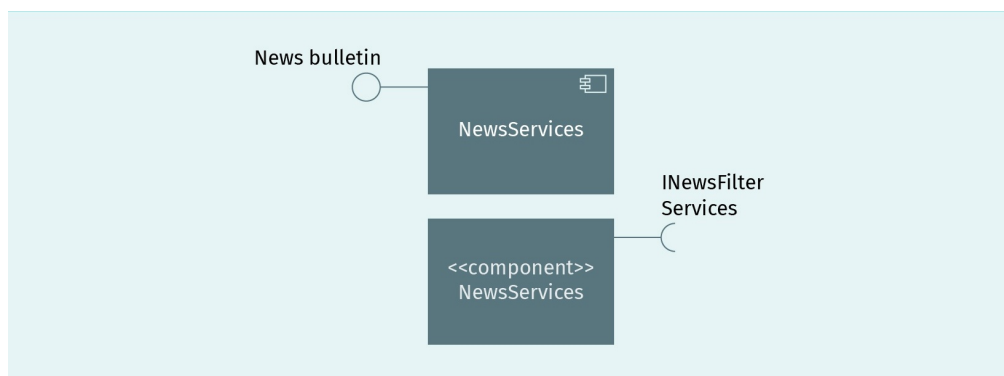
Figure 10: Example of Interface Usage



Source: Venter, 2021.

A component is a part of a system that conforms to a set of interfaces. It also provides realization to interfaces; it is physical and replaceable, and typically represents the physical packaging of logical elements. A component manifests using interfaces, and may be manifested by one or more artifacts. It is drawn as a classifier rectangle containing the keyword <<component>> and can also be shown as a rectangle with an icon made up of a small rectangle with tabs in the top right corner; every representation includes its name. In the example below, the component “NewsServices” manifests because of, for example, a news bulletin (i.e., news that is being broadcasted through the interface “News bulletin”). The following example shows both notations as well as interfaces.

Figure 11: Examples of Components and Interfaces

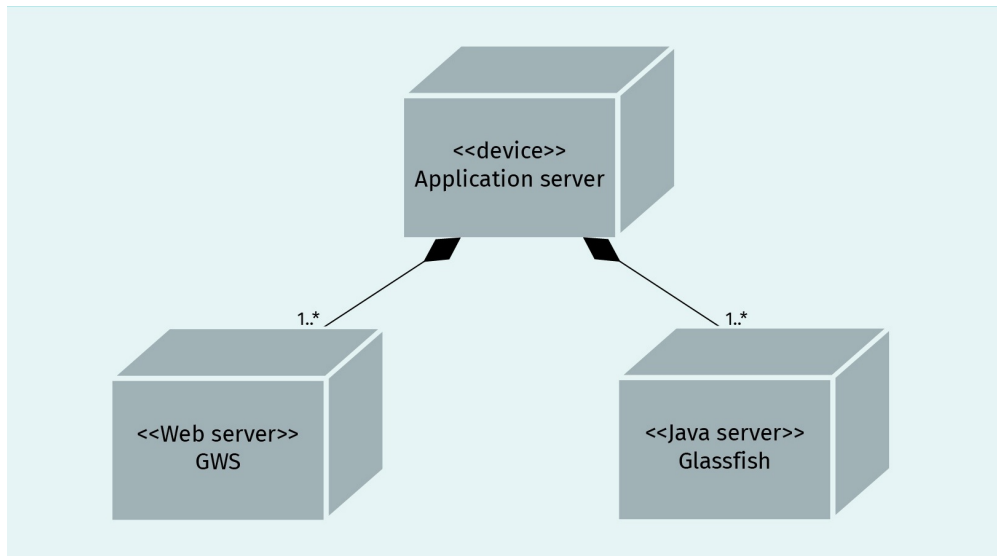


Source: Venter, 2021.

Components use two types of interfaces: provided and required. A provided interface implements the component. In the example above, the interface “News bulletin” implements the component “NewsServices”, when a news bulletin is being broadcasted. A required interface is one that the component requires to manifest. The example above shows the component NewsServices that requires the interface NewsFilter (e.g., to provide current news, news must be filtered so that only current news is aired).

A node exists in run-time, representing a computational resource that typically has memory, processing capability, or both. The node is associated with deployments of artifacts; for example, artifacts may be deployed upon nodes for execution. Nodes can be connected via communication paths, which can also be defined between nodes. Links between node instances will then define specific network topologies. A node is drawn as a cube and includes its name. A hierarchical node showing an application server running several web and Java servers is shown below.

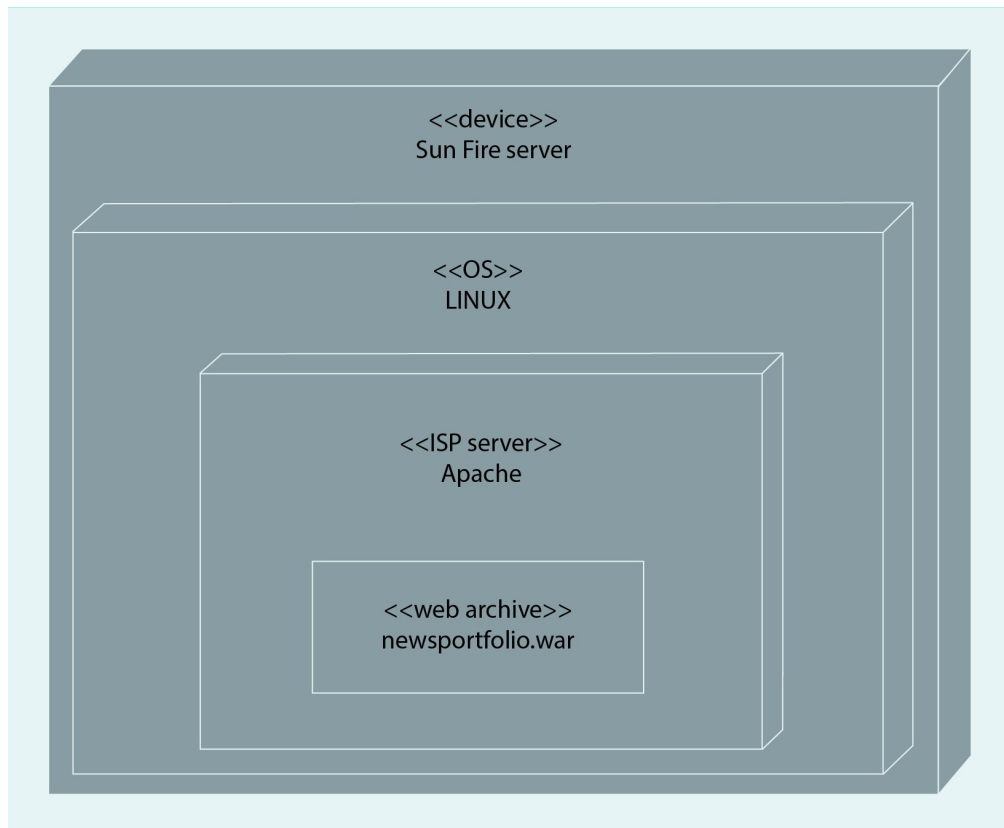
Figure 12: Example of a Hierarchical Node



Source: Venter, 2021.

An execution environment is a node that offers an environment in which a component can execute. This environment is for particular types of components, deployed as executable artifacts. It is typically part of a general node or <<device>>, representing a physical hardware environment; it can be nested, as in the example below.

Figure 13: Example of Execution Environments Nested into a Server Device



Source: Venter, 2021.

Behavioral things and abstractions, representing behavior over space and time, include interactions and state machines. A state machine specifies the sequences of states that an object or interaction undergoes during its lifetime, including its responses to the events. It is drawn as a rounded rectangle and typically includes its name and sub-states (if any). State machines can represent behavioral states or protocol states. Behavioral states specify discrete behavior, e.g., an electronic banking system waiting for a customer to transact. It may also include labels, such as activity labels to indicate behavior to perform upon entry during, or when exiting the state. The behavioral state machine “Waiting”, with activity labels “Entry” and “Exit”, is shown in the example below.

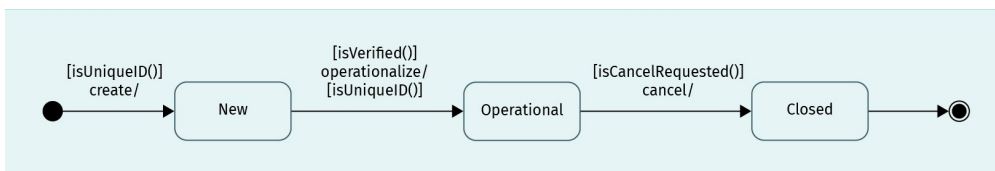
Figure 14: Example of a Simple State Behavioral State Machine



Source: Venter, 2021.

Protocol states in state machines present the external view. An interaction comprises a set of messages that are exchanged among a set of objects—it is done within a specific context and for a specific purpose. It is drawn as a direct line and includes the name of the operation. A diagram containing protocol state machines (simple state) with protocol transitions is shown below.

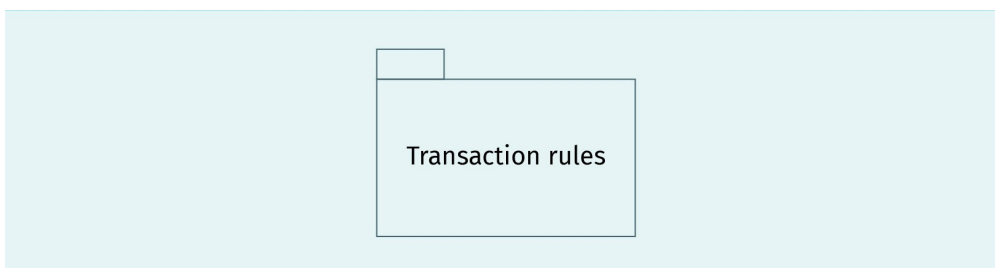
Figure 15: Example of Protocol State Machines and Protocol Transactions



Source: Venter, 2021.

Boxes (referred to as packages), are used to group things and abstractions; models can be grouped into packages. A package is a general-purpose, purely conceptual, mechanism that is used to organize and group elements. Variations of packages are frameworks, models, and sub-systems. They are drawn as tabbed folders and typically include only their names, but can also include their contents. A package “Transaction rules” is shown below.

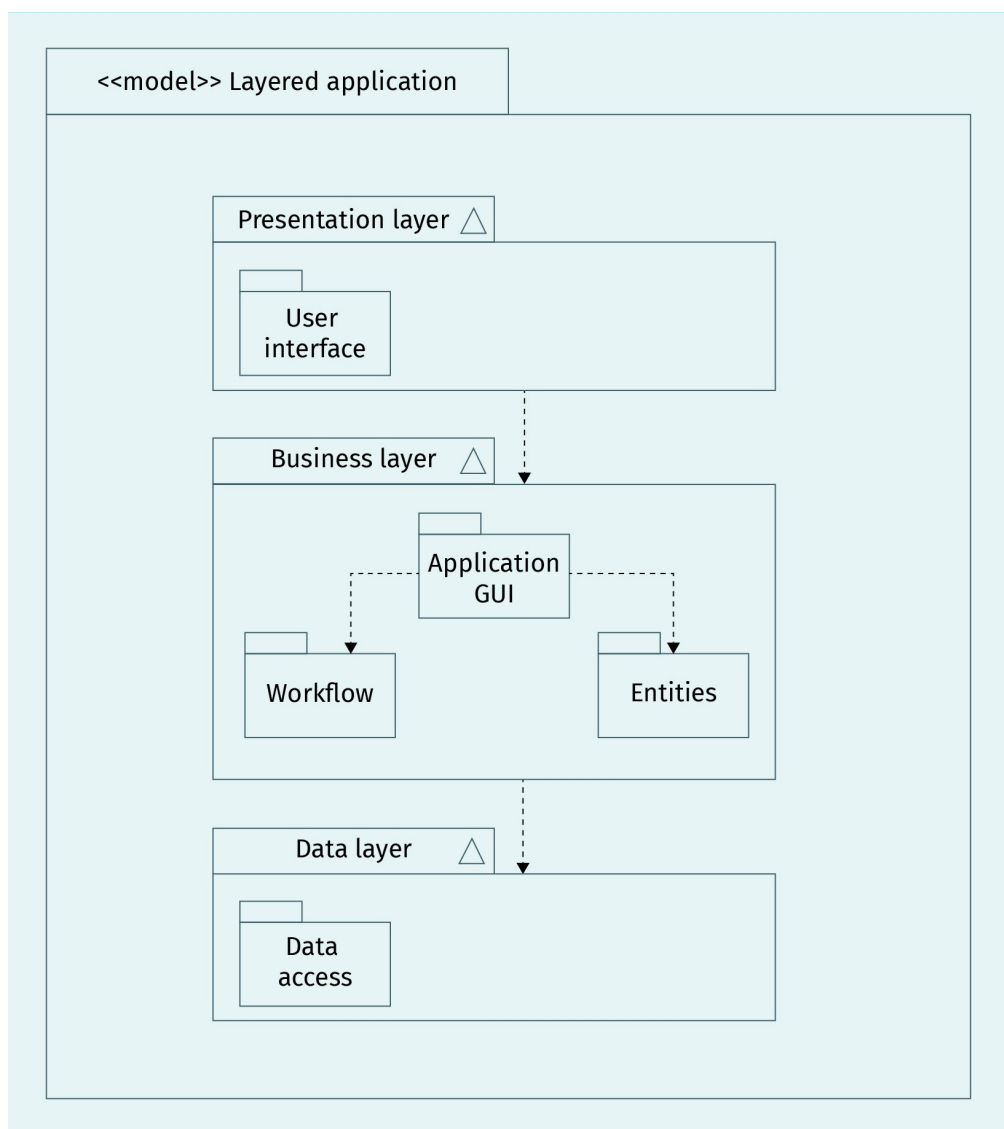
Figure 16: Example of a Package



Source: Venter, 2021.

The following figure is a model diagram that includes a “container” model (“Layered application”). It contains three other models, and their associated packages and dependencies are indicated.

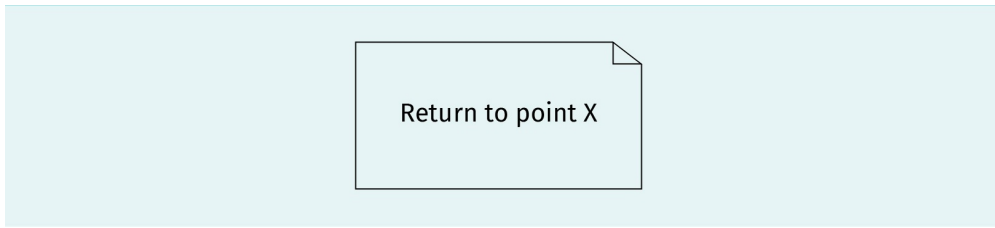
Figure 17: Example of a Model Diagram



Source: Venter, 2021.

Annotational things and abstractions are comments that describe and illuminate elements in a model; in other words, they are notes. A note is a symbol used to render constraints and comments attached to an element or collection of elements. It is drawn as a rectangle with a dog-eared corner and includes a textual or graphical comment as shown below.

Figure 18: Example of a Note



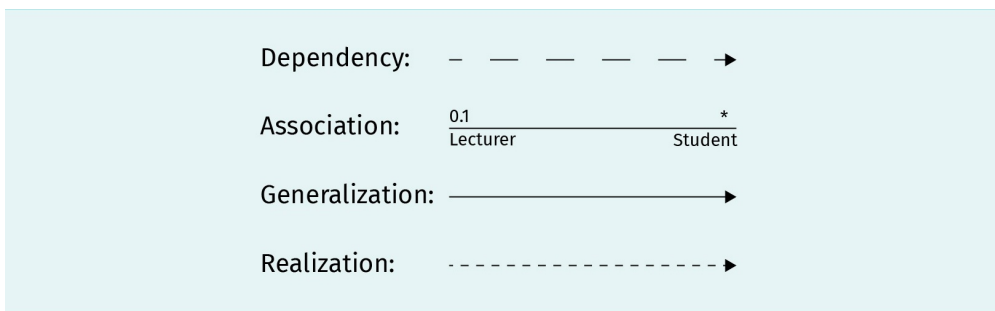
Source: Venter, 2021.

The basic forms of relationships are based on either dependency, association, generalization, or realization. A dependency denotes a semantic relationship where the independent thing affects the semantics of the dependent thing. It is drawn as a dashed line, and may include a label. An association is a structural relationship that describes a set of **links**. It is drawn as a solid line, and may include a label and other association adornments, e.g., aggregation, role names, and multiplicity. In the example below, its multiplicity indicates that zero to infinity (denoted with the asterisk *) students can be associated with between zero and one lecturers. A generalization shows a relationship where the specialized (child) element's objects are substitutable for those of the generalized (parent) element, i.e., the child object shares the parent object's structure and behavior. It is drawn as a solid line with an arrowhead pointing in the direction of the parent. A realization is a semantic relationship between classifiers; one classifier specifies a contract that another guarantees to **carry out**. They occur between interfaces and the classes or components that realize them, and also between use cases and the collaborations that realize them. In addition to these **basic relational things**, the following variations can occur:

Link
A link is a connection between objects.

- refinement
- trace
- include
- extend (for dependencies)

Figure 19: Examples of Basic Relationships



Source: Venter, 2021.

Rules in UML

Rules ensure that models are **well-formed**, i.e., that they are semantically self-consistent and harmonious with related models. Models follow the semantic rules for

- names, which refer to UML **things**, relationships, and diagrams.
- scope, which refers to the context and gives specific meaning to a name.
- visibility, which refers to how names are understood and used.
- integrity, which refers to how things consistently relate to each other.
- execution, which refers to the manner in which a dynamic model is run or simulated.

Since models tend to evolve, and are typically viewed by many stakeholders in various ways and at different times, models can also be built to be elided, i.e., with certain elements hidden for a simplified view, incomplete, missing certain elements, or inconsistent (not guaranteeing the integrity of the model). Hence, to satisfy diverse stakeholder groups, compromises are made between the level of abstraction versus the amount of detail. These models may also naturally evolve and mature over time as more information becomes available.

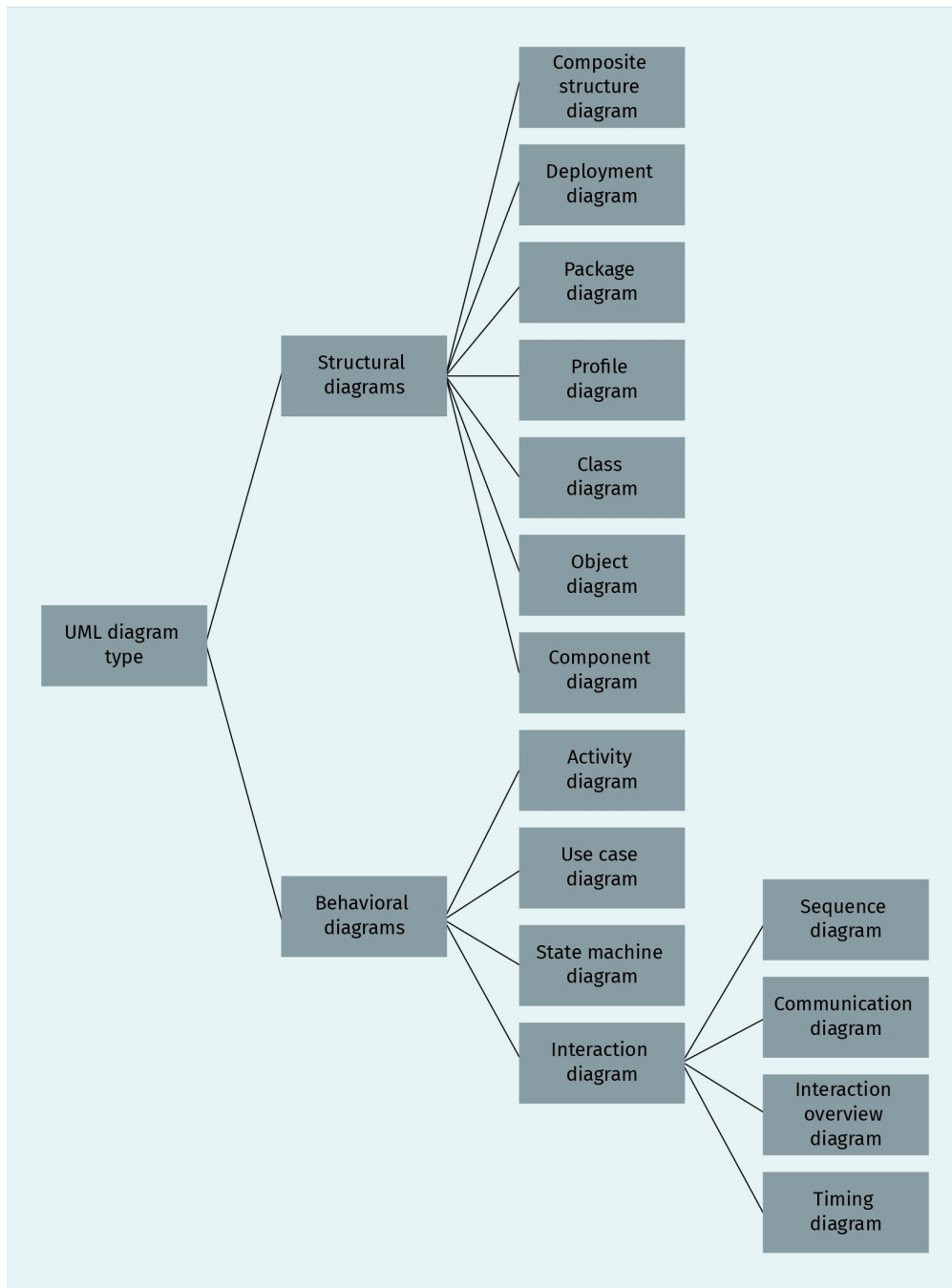
Common mechanisms in UML

Four common mechanisms are applied consistently. These are: specifications, adornments, common divisions, and extensibility mechanisms. Specifications provide textual statements related to the building blocks' syntax and semantics, providing all the attributes, operations, and behaviors the class embodies. Adornments indicate, e.g., whether it is abstract and the visibility of attributes and operations. Common divisions distinguish between classes and objects, as well as between an interface, declaring a contract and its implementation, and representing a concrete realization of said contract. Extensibility mechanisms include stereotypes, tagged values, and constraints. A stereotype extends the UML vocabulary with terms derived from, e.g., the applicable programming language. A tagged value facilitates addition of new and relevant information as applicable, and a constraint allows for addition of new and modification of existing rules.

UML diagrams

A UML diagram graphically presents a set of elements. These diagrams are used to visualize a system from various perspectives and are typically drawn as connected graphs of vertices (things) and arcs (relationships). In UML, one can differentiate between different types of diagrams. The main categories of diagrams are static structural diagrams and dynamic behavioral diagrams. They are illustrated below.

Figure 20: UML Diagram Types



Source: Brückmann, 2013.

Structural diagrams illustrate a system's static structure, elements, composition, and interfaces; they are composite structure diagrams, deployment diagrams, profile diagrams, class diagrams, object diagrams, and component diagrams. Behavior diagrams indicate a system's flow of activities and interactions; these are as follows:

- activity diagrams
- use case diagrams
- state machine diagrams and interaction diagrams
- communication diagrams
- sequence diagrams
- interaction overview diagrams
- timing diagrams

In practice, diagrams are used selectively and on a case-by-case basis, as they are applicable and useful for the context to be modeled, as well as for the stakeholders involved. The table below indicates static structure and dynamic behavior diagrams that are often used in UML.

Table 1: UML Static Structure and Dynamic Behavior Diagrams

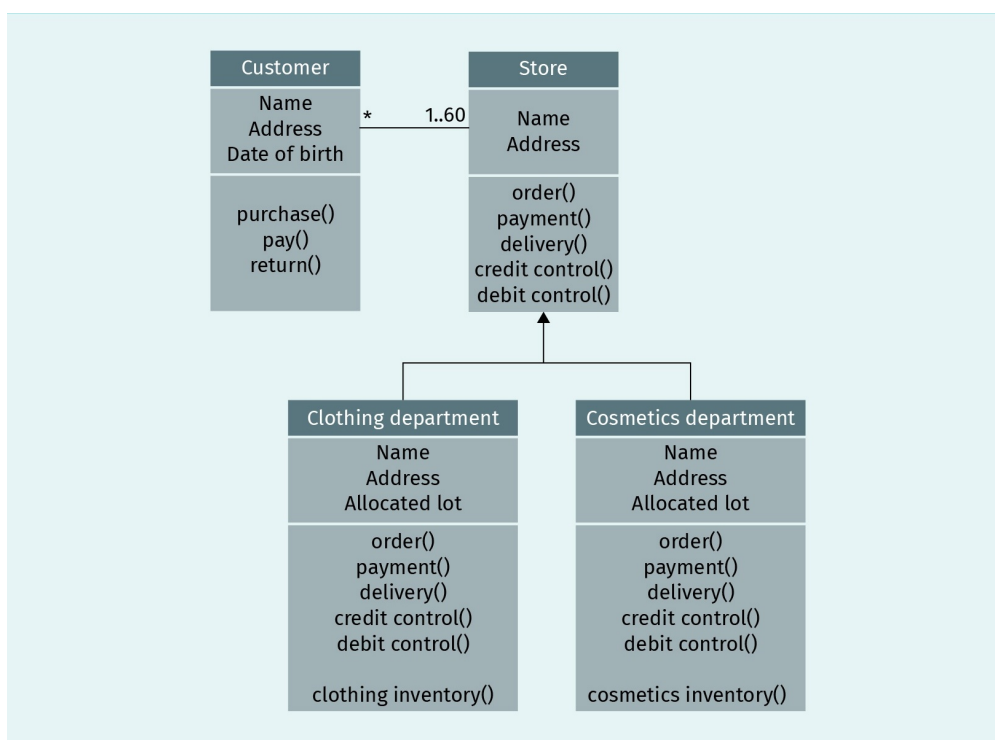
Diagram	What does this diagram address?
Class diagram	<ul style="list-style-type: none"> • a system's static design and structural view • shows a set of classes, interfaces, operations, collaborations, and relationships
Object diagram	<ul style="list-style-type: none"> • static snapshots of instances of the things in class diagrams • shows a set of objects and their relationships
Component diagram	<ul style="list-style-type: none"> • the static implementation view of a system • shows the organization and dependencies among a set of components
Deployment diagram	<ul style="list-style-type: none"> • an architecture's static deployment view • shows configuration of run-time processing nodes and the components living on them
Use case diagram	<ul style="list-style-type: none"> • a system's static use case view • shows use cases, actors, and relationships indicating functionality from a user (and external) view
Communication diagram	<ul style="list-style-type: none"> • a system's dynamic interaction view • shows objects and their relationships, focusing on their topology
Sequence diagram	<ul style="list-style-type: none"> • a system's dynamic interaction view • shows objects and their relationships, including messages being dispatched
Activity diagram	<ul style="list-style-type: none"> • a system's dynamic view • highlights the flow of control among objects and actors • shows interactions and flows between activities within a system
Collaboration diagram	<ul style="list-style-type: none"> • structural organization of objects sending or receiving messages

Diagram	What does this diagram address?
State machine diagram	<ul style="list-style-type: none"> • a system's dynamic behavioral view • emphasizes an object's event-ordered behavior • shows a state machine that consists of states, transitions, events, and activities models behavior of an interface, class, or collaboration

Source: Venter, 2021.

The class diagram is one of the most frequently used diagrams; it models classes with their associated attributes, methods, and relationships. The other types of UML diagrams are essentially based on the modeling concepts of the class diagram. The structures in which information is stored within the relevant IT systems are made visible by class diagrams. Below is an example that shows relevant associations between these classes. In this diagram, its multiplicity indicates that an infinite number (denoted by the asterisk *) of customers can visit one of 60 (denoted by 1..60) available branches of a store. The superclass "Store" is indicated as having a single-inheritance hierarchy, meaning that the two subclasses (the "Clothing department" and the "Cosmetics department") inherit all the features from only one superclass, "Store", and have more features of their own.

Figure 21: Examples of a Class Diagram

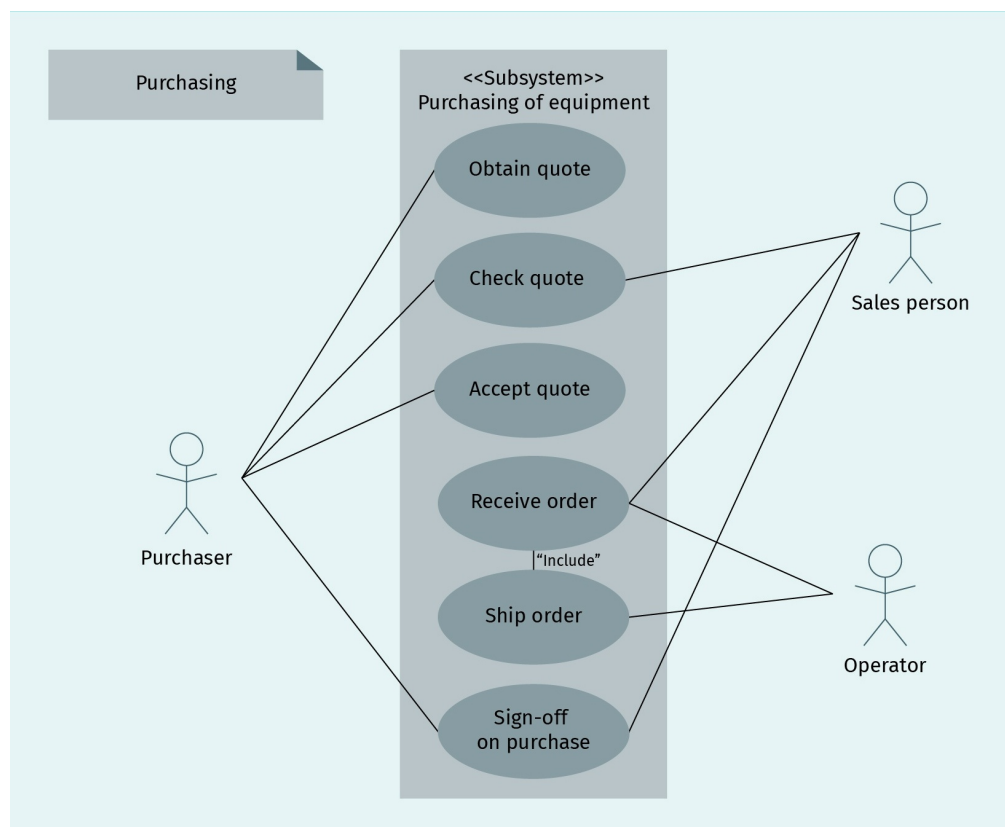


Source: Venter, 2021.

A use case diagram describes a system's functionality from the perspective of the users (also referred to as actors in UML). They do not describe procedures, but are non-technical diagrams that show, e.g., an overview of a business process. They show associations or

interactions (using lines) between named actors (indicated by stick figures or any suitable symbol indicating applicable characteristics) and relevant use cases (indicated by ellipses). These interactions are from the perspective of the user and therefore also entail a view of the user requirements. In terms of the software systems, use case diagrams give an external view. An example of a use case diagram for a portion of a business system that entails purchasing (i.e., “Purchasing of equipment”) is shown below.

Figure 22: Example of a Use Case Diagram



Source: Venter, 2021.

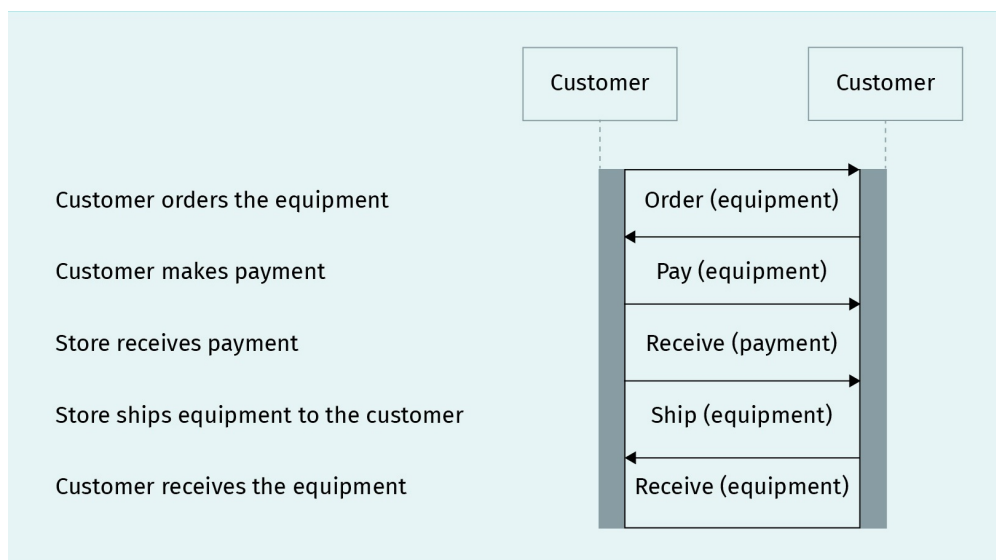
In this example, the involved actors are a “Purchaser”, a “Sales person”, and an “Operator”. They are involved in purchasing equipment (as the depicted business process), so the focus of this diagram is on the process of purchasing equipment, hence, the subject is indicated in the diagram as “Purchasing of equipment”. It entails the following use cases “Obtain quote”, “Check quote”, “Accept quote”, “Receive order”, “Ship order”, and “Sign-off on purchase”. The “Include” relationship, indicated between the use cases “Receive order” and “Ship order”, signifies that receiving an order also includes shipping an order. To summarize, the purchaser will obtain a quote, check it, accept it, and finally sign-off on the purchase made. The salesperson will also be involved in the functionalities to check quotes, receive orders, and sign-off on purchases. The operator will be involved to receive order information and ship orders. The sequence of these is not necessarily depicted in chronological order, but merely show the required functionalities and actors’ involve-

ments. The actors in use case diagrams can be humans (as in the example above), but they can also be non-human and subsequently be depicted by class diagrams. For example, an actor can be a piece of hardware, software, or another system.

Following on from use case diagrams, sequence diagrams show the sequenced interactions between the objects and messages. They are used to expand the detail of a use case, and show the lifelines of objects, i.e., they help to visualize and specify the flow of control and they indicate, in a visual and simplified way, when objects should start, iterate, and cease to exist. Sequence diagrams describe a chain of chronological interactions; they indicate data and information that are exchanged between actors. They are widely used for their simplicity and ease of use as they require few graphical elements and are easy to read. Similarly, communication diagrams show the relationships between objects, but the focus is on their topology. Sequence diagrams and communication diagrams essentially show what happens in a system when a user accesses and uses it. In practice, sequence diagrams are used more often than communication diagrams (Baumann et al., 2005).

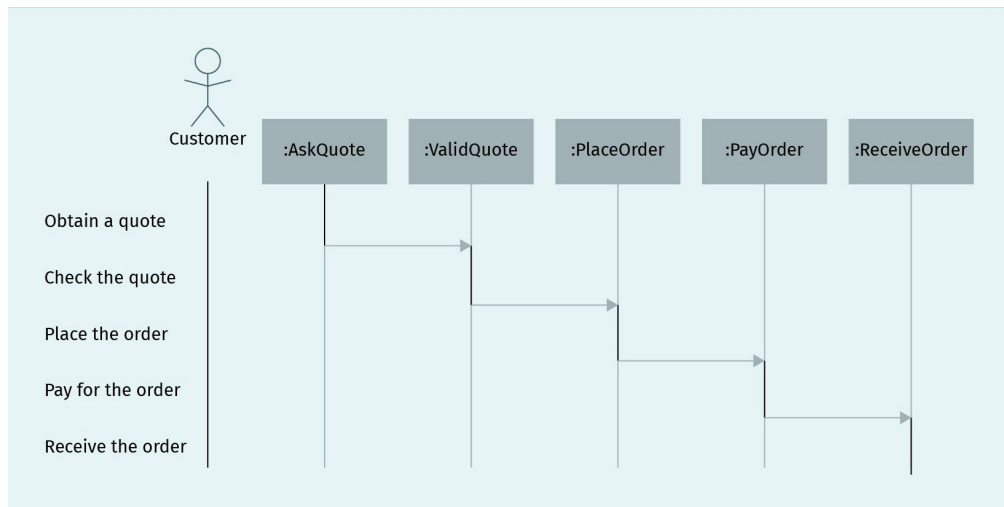
UML allows for variations of diagrams. In the top diagram of the figure below, a basic high-level sequence diagram shows an object “Customer” that is involved to order, pay for, and receive equipment from an object “Store”, which receives the payment and ships the equipment. The diagram is annotated with comments, e.g., indicating that the “Customer orders the equipment”. The messages are depicted in increasing chronological order, from the top to the bottom, with the direction of the arrows indicating the direction in which the message is being sent. In the second diagram of the figure below, a basic high-level sequence diagram is illustrated. It shows what happens when the object “Customer” obtains a quote and places an order. The diagrams below are simplified examples that do not indicate what happens if any errors are encountered.

Figure 23: Example of a Sequence Diagram (a)



Source: Venter, 2021.

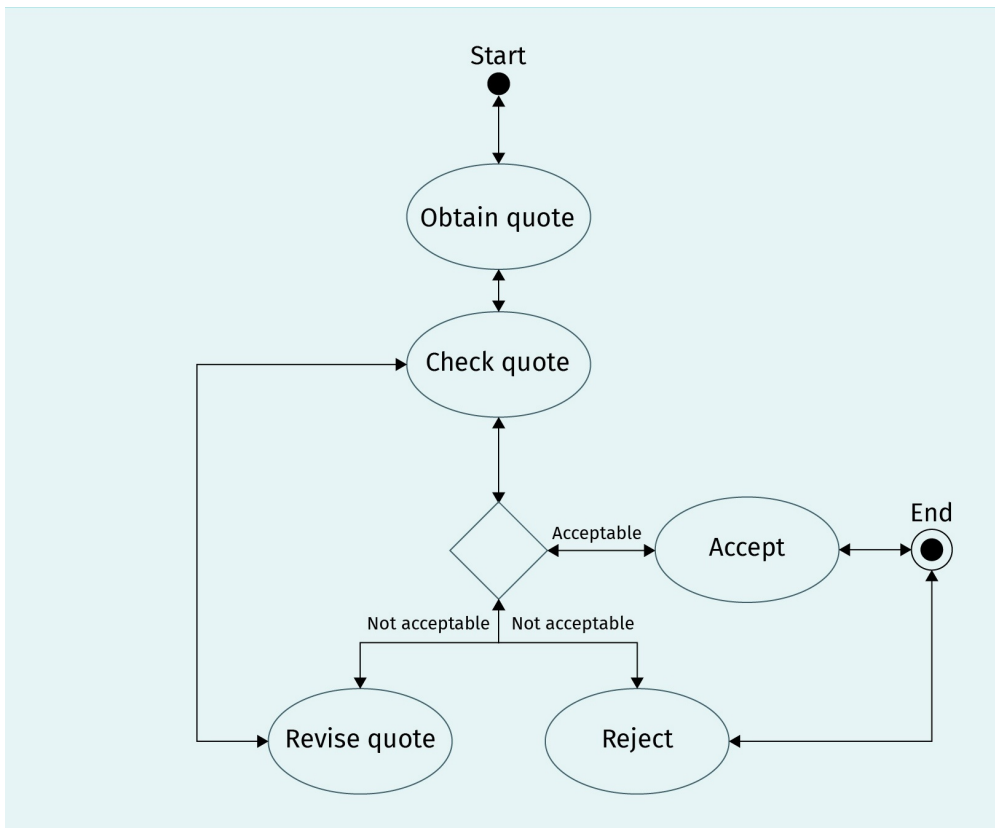
Figure 24: Example of a Sequence Diagram (b)



Source: Venter, 2021.

Activity diagrams describe the procedures that make up a process; they are related to flowcharts and illustrate the chronological and parallel order of activities, and relevant alternatives. They facilitate functional thinking, rather than object-oriented thinking, and are therefore particularly useful when exploring and illustrating business processes (Baumann et al., 2005). Below is an example of an elementary high-level activity diagram indicating the procedures involved to receive, accept, and reject a quote.

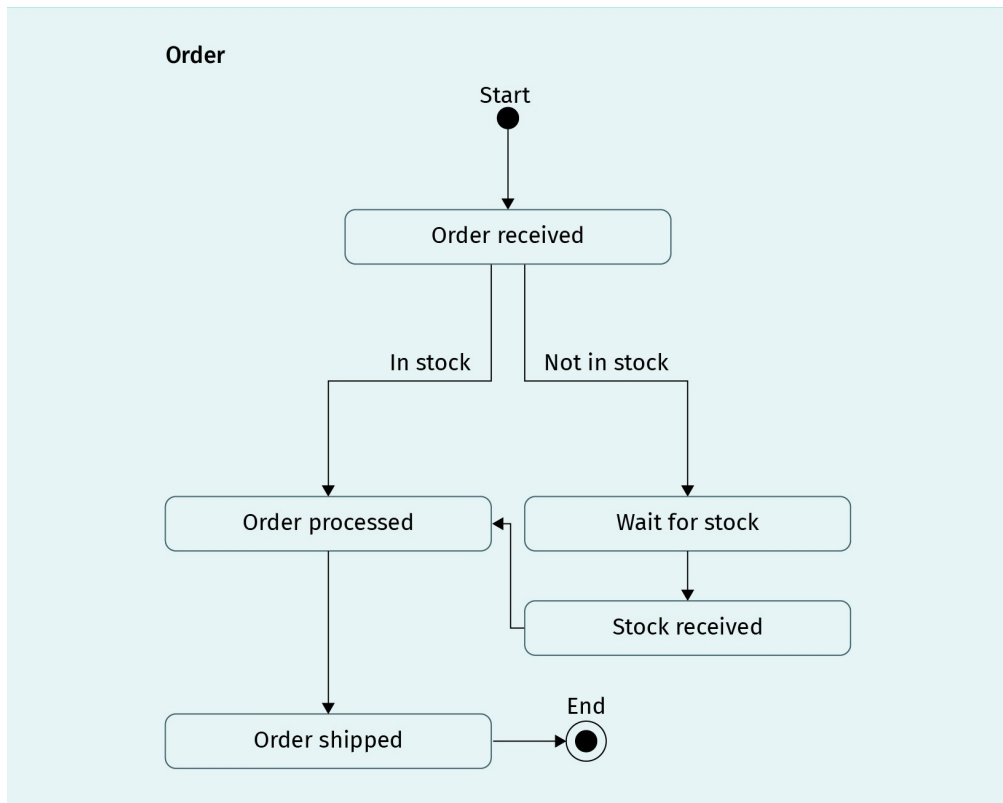
Figure 25: Example of an Activity Diagram



Source: Venter, 2021.

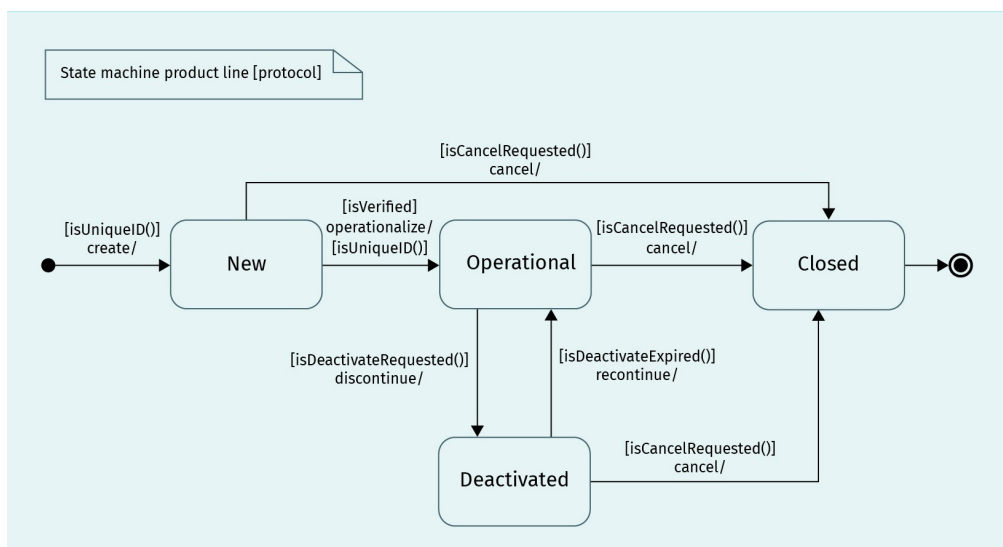
State machine diagrams illustrate the different legitimate states in which an object may be; they also indicate the states that objects may transfer to when events occur and when they receive messages. The “state” of these diagrams refers to a specific set of values that the attributes of the object have at a specific time. The object’s state changes when the values of the attributes change. Activity diagrams initially originated as a variation of state machine diagrams that focuses on internal flows and activities of an object, a set of objects, or an entire use case. Activity diagrams were re-formalized in the latest version of UML and are now based on semantics that are similar to that of Petri nets, thus increasing the scope of situations with which it can be modeled. State machine diagrams give either a behavioral view, meaning that they show everything that can happen with or to an object or a protocol (external) view. The following figure shows an example of a simple behavior state machine diagram for the object “Order” and a protocol state machine diagram for indicating whether a “Product line” is operational or inactive (deactivated).

Figure 26: Example of a State Machine Diagram (a)



Source: Venter, 2021.

Figure 27: Example of a State Machine Diagram (b)



Source: Venter, 2021.

Systems Modeling Language

Systems Modeling Language (SysML) is a modeling language that originated in 2001; it is a dialect of UML and defined by an UML profile. SysML supports model-based systems engineering (MBSE), is used to model large and complex systems engineering applications, and supports many different systems engineering methods (Friedenthal et al., 2012). It supports the specification, analysis, design, verification, and validation of systems as well as interconnected systems-of-systems. Revolutionary IT and its growing networked interconnectedness with a myriad of other products, devices, and services continues to bring about software and software systems that do not exist in isolation. These must be explored, visualized, and modeled as part of the bigger system-of-systems. Derived from UML, SysML is systems-centric (rather than software-centric) and allows a broad range of systems to be modeled. Its modeling range includes software-intensive systems, but is not limited to it.

SysML is regarded as flexible and expressive, and is relatively easy to learn and to apply. It is aligned with the standard IEEE-Std-1471-2000, i.e., the *IEEE Recommended Practice for Architectural Description of Software Intensive Systems* (IEEE, 2000). Friedenthal et al. (2012) explain that SysML concepts involve the following three parts:

1. an abstract syntax or schema that defines the language concepts and is described by means of a meta model
2. a concrete syntax or notation that defines how language concepts are represented and is described by means of notation tables
3. semantics or meaning that gives the meaning of the language concepts

In SysML, UML is extended to include the following system modeling capabilities:

- model elements,
- requirements,
- blocks,
- activities,
- constraint blocks,
- ports and flows, and
- allocations.

Accordingly, SysML diagrams include package diagrams, requirement diagrams, behavior diagrams, parametric diagrams, and structure diagrams. Similar to UML, behavior diagrams include activity diagrams, sequence diagrams, state machine diagrams, and use case diagrams. Structural diagrams for SysML include block definition diagrams and internal block diagrams.

Friedenthal et al. (2012) say that a block is “the principle structural construct of SysML” (p. 119) and define it as “the modular unit of structure...used to define a type of system, component, or item that flows through the system, as well as external entities, conceptual entities or other logical abstractions...describes a set of uniquely identifiable instances that share the block’s definition...is defined by the features it owns, which may be subdivided into structural features and behavioral features” (p. 119). A block is a representation

of a conceptual (logical) or physical entity or object. It can describe components that are reusable across various systems. A block is similar to an object-oriented class in that it describes a set of similar instances or objects that share common characteristics. The table below defines structural (block) diagrams.

Table 2: SysML Structural Diagrams

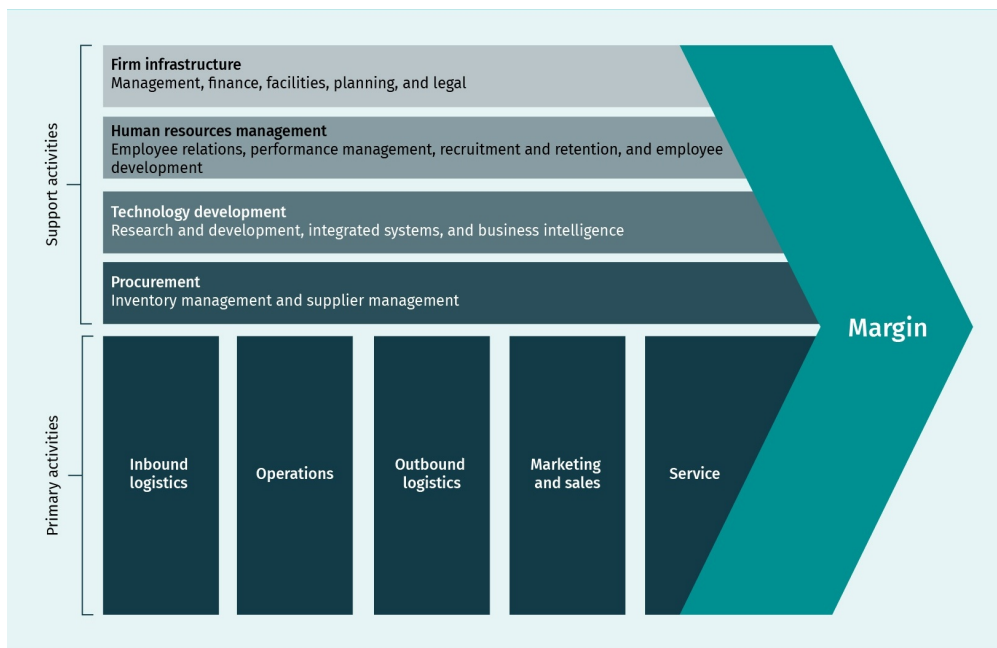
Diagram	What does this diagram address?
Block definition diagram	definition of blocks in terms of features, as well as structural relationships with other blocks
Internal block diagram	illustration of connections between the parts of a block

Source: Venter, 2021.

2.2 Notations for Modeling the Interaction between Processes

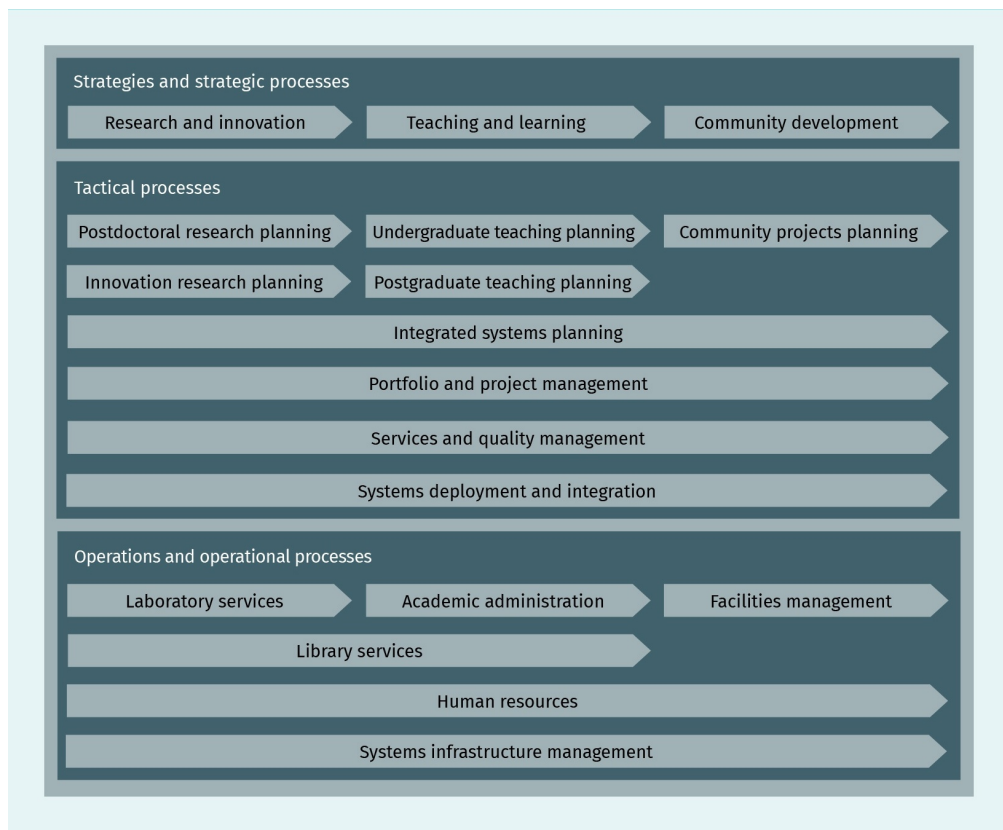
Visualizations (e.g., diagrams) that show an organization's value chains and process landscapes are relevant to illuminate the organization's outward and internal view, respectively (Kneuper, 2018). According to Porter and Millar (1985), a value chain comprises of an organization's activities, as well as its linked interactions and interdependencies. It entails primary activities, i.e., inbound logistics, operations, outbound logistics, marketing and sales, and service, as well as support activities, i.e., firm infrastructure, human resources management, technology development, and procurement. The goal of these activities is to create value in excess of the cost to conduct and obtain them, i.e., to generate maximum profit. IT, such as software systems, has strategic significance and implications for modern-day companies, regardless of the company's core business. The effects thereof must be considered, analyzed, and understood as an intrinsic part of the value chain. A process landscape, on the other hand, gives an internal view of an organization; it shows the interconnected roles of certain internal and software processes. These must also be understood by the software engineer in order to comprehend the complexity of integration where it is relevant. Ultimately, all of these elements will significantly impact the enterprise architecture framework and determine business requirements for software systems, as well as for the governance and management of software processes. The examples below illustrate a high-level value chain and a simplified process landscape.

Figure 28: A Value Chain



Source: Venter, 2021.

Figure 29: A Process Landscape



Source: Venter, 2021.

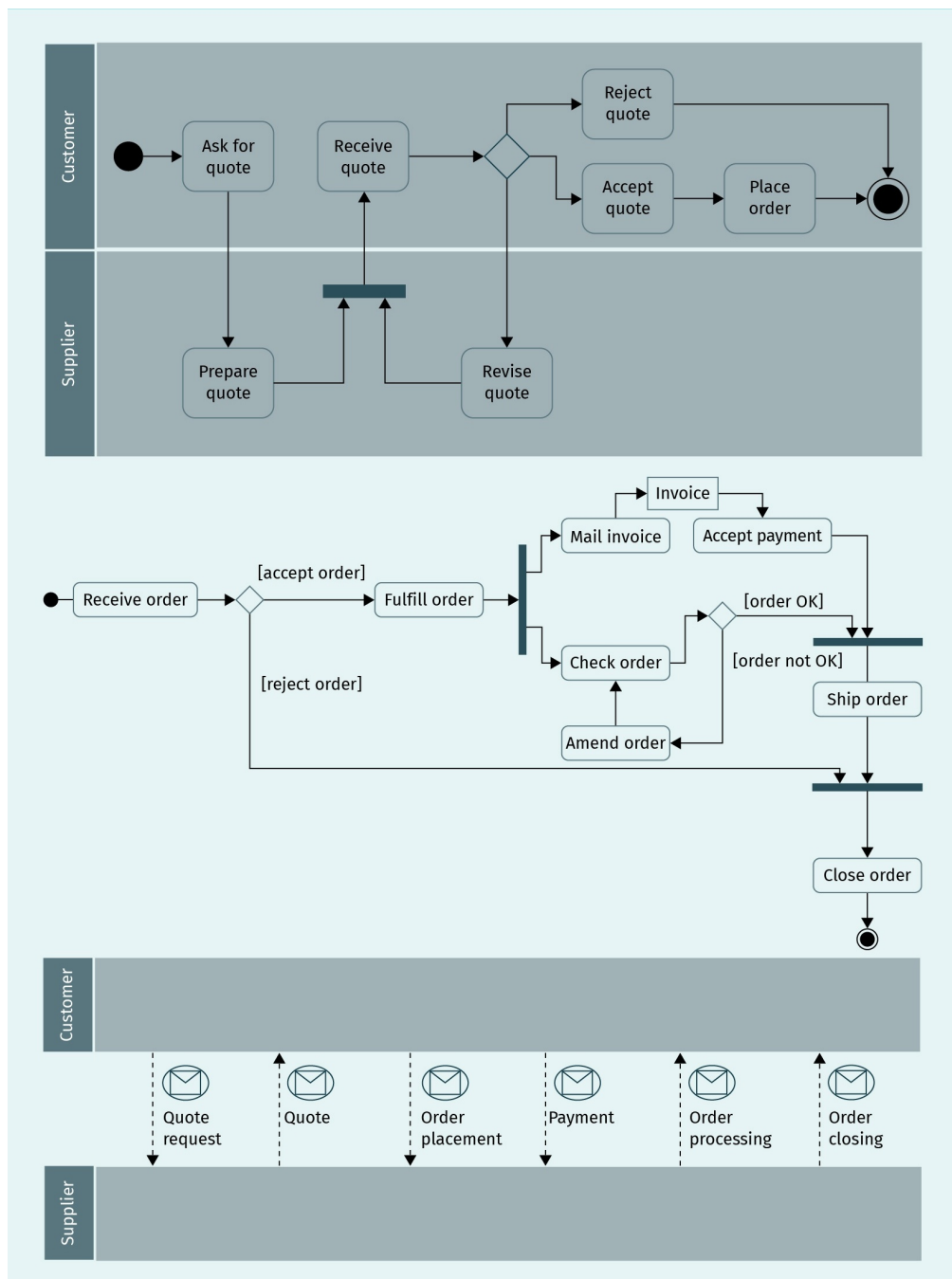
The Multi-View Process Modeling Language (MVP-L) is used to formally describe processes, in terms of external behavior, without considering internal structure. MVP-L originated in the 1980s at the University of Maryland, USA, and continued to be refined at the *Universität Kaiserslautern*, Germany. Rombach and Verlage (1993) explain that “MVP focuses on process models, their representation, their modularization according to views” and that “MVP-L was designed to help build descriptive process models, package them for reuse, integrate them into prescriptive project plans, analyze project plans, and use these project plans to guide future projects” (p. 154). MVP-L uses processes, products, resources, and quality attributes, and applies instantiations of these project plans. Münch et al. (2012) explain that processes refer to the activities performed during a project that produce, consume, or modify products. Products are the resultant software products of the development or maintenance processes, including the final software, documentation, etc. Models entail the activities performed during a project that produces, consumes, or modifies products. Attributes refer to defined measurable properties of products, resources, and processes.

All of the diagrams presented in the first and third section of this unit can also be used to model interactions between processes and connected segments of larger processes. As an example, both activity diagrams and BPMN represent interactions between single steps, activities, or tasks. Complex processes and processes addressing different and diverse

parts of a business and system can be modeled by dividing them into sub-processes (or segments) and modeling these individually to visualize the detail, then combining them again using one overview diagram.

The example below shows three diagrams. The first diagram illustrates the sub-process (these are fragments or segments of a bigger business process) followed to obtain a quote for an order to be placed. The second diagram illustrates the ensuing sub-process that details the activities when an order is placed; this diagram also shows that an activity diagram can include an object (i.e., “Invoice”). In the first activity, diagram partitions are depicted, using swim lane notation and indicating interactions between the stakeholders—a customer and supplier. The final diagram gives a high-level overview where these are combined to show the overall purchasing process using BPMN notation.

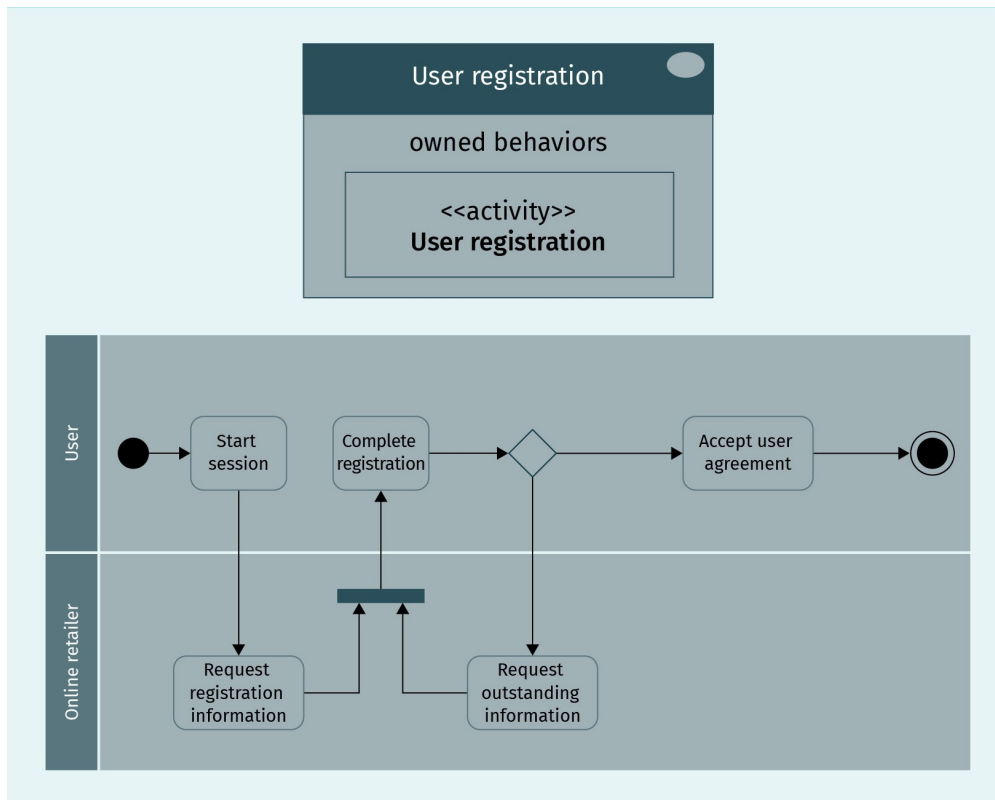
Figure 30: Modeling of Linked Subprocesses and a High-Level Overview



Source: Venter, 2021.

Moreover, behaviors of use cases can be described using natural language text or different UML diagrams. The example below shows that the use case “User registration” owns the behavior represented by the activity of “User registration”. This illustrates the binding of a use case to an activity. The activity diagram that follows shows the behavior of the activity to register a user with an online retailer.

Figure 31: Binding of a Use Case to an Activity



Source: Venter, 2021.

2.3 Detailed Level Notations

Kneuper (2018) argues that different notations can be used to describe detailed levels of individual processes. He notes that many of these are not specifically for software processes since they originate from business processes and other areas of application and support the modeling of multiple processes as well as hierarchical modeling with increasing levels of detail. They include, e.g., process patterns, modeling notations from requirements analysis, high-level notations for general processes, notations for modeling business processes, and process notations for formal analysis. Reasons for selecting a modeling notation typically include general properties, such as how easy it is to understand and use, and how expressive it is.

Process patterns describe tasks, stages, and phases from the bottom-up, and with reference to relevant activities, actions, products, and behaviors to solve a problem. It can include, e.g., the problem, context, solution, typical roles, and artifacts, as in the approach presented by Neatby-Smith (1999). In this approach, task patterns define steps that must be executed to complete tasks. Stage process patterns include several task process patterns that are required to be completed in order to move to a next stage, and phase patterns are comprised of two or more stage patterns. Furthermore, input and output arti-

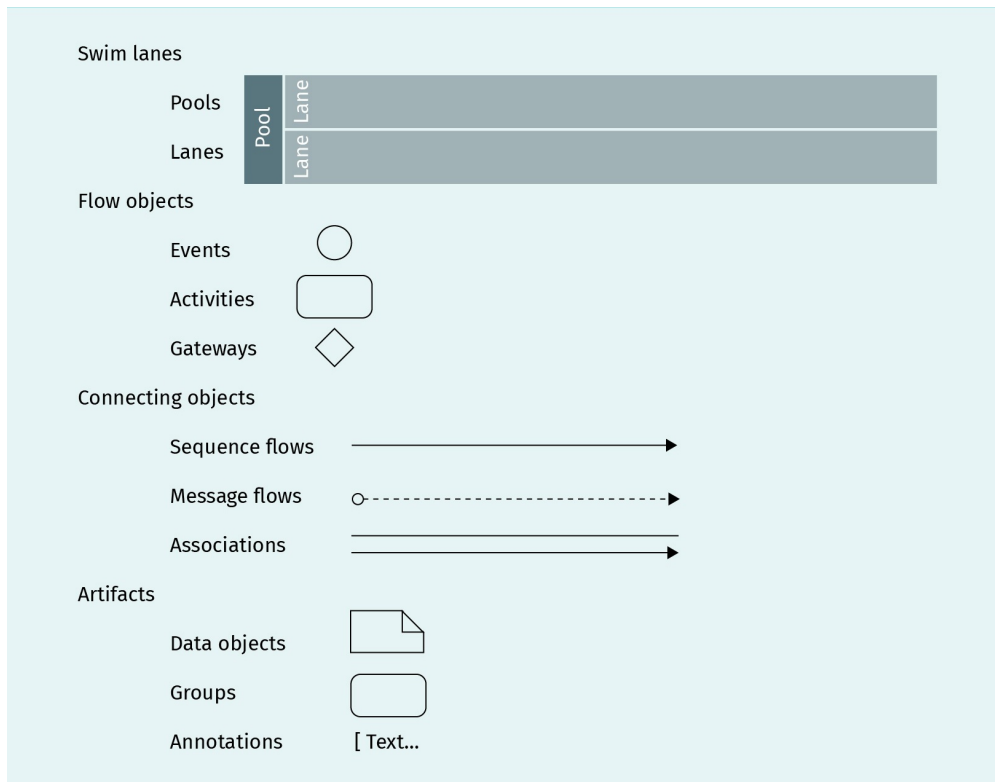
facts, as well as roles, are assigned for each pattern. In contrast to task process patterns and phase patterns, only stage process patterns follow an actual pattern. Stage process patterns are initial context, solution, project tasks, resulting context, secrets of success, and process checklist.

Another modeling notation that is particularly relevant, and widely used as part of software engineering, is the **Business Process Model and Notation (BPMN)**. It is based on business process management and modeling principles (Mathias, 2019). It was originally developed by the Business Process Management Initiative (BPMI) and is now also maintained by the Object Management Group (OMG) (these organizations merged in 2005). BPMN remains, regardless of some drawbacks and issues, one of the most widely applied notations, and is used by both software architects and business analysts (Kossak et al., 2014). The ISO standard for BPMN is ISO/IEC 19510:2013, which provides an easily understandable and readable notation and is accordingly suitable for both technical and non-technical users and audiences (International Organization for Standardization, 2013). BPMN diagrams describe a similar type of activity as UML activity diagrams, but they use a different notation. It is process-oriented, making it useful in the business process domain, and hence used to illuminate a business process. BPMN is an expressive notation that describes the flow of activities within a process. Diagrams are graphical and deliberately kept simple to ease understanding and readability. They typically employ the following elements:

- flow objects, i.e., events, activities, and gateways;
- connecting objects, i.e., sequence flows, message flows, and associations;
- swim lanes, i.e., pools and lanes; and
- artifacts, i.e., data objects, groups, and annotations.

A BPMN diagram can also contain other illustrations, e.g., documents, different types of catching and throwing messages, compensations, timers, errors, signals, and links. A selection of BPMN elements are illustrated in the following figure.

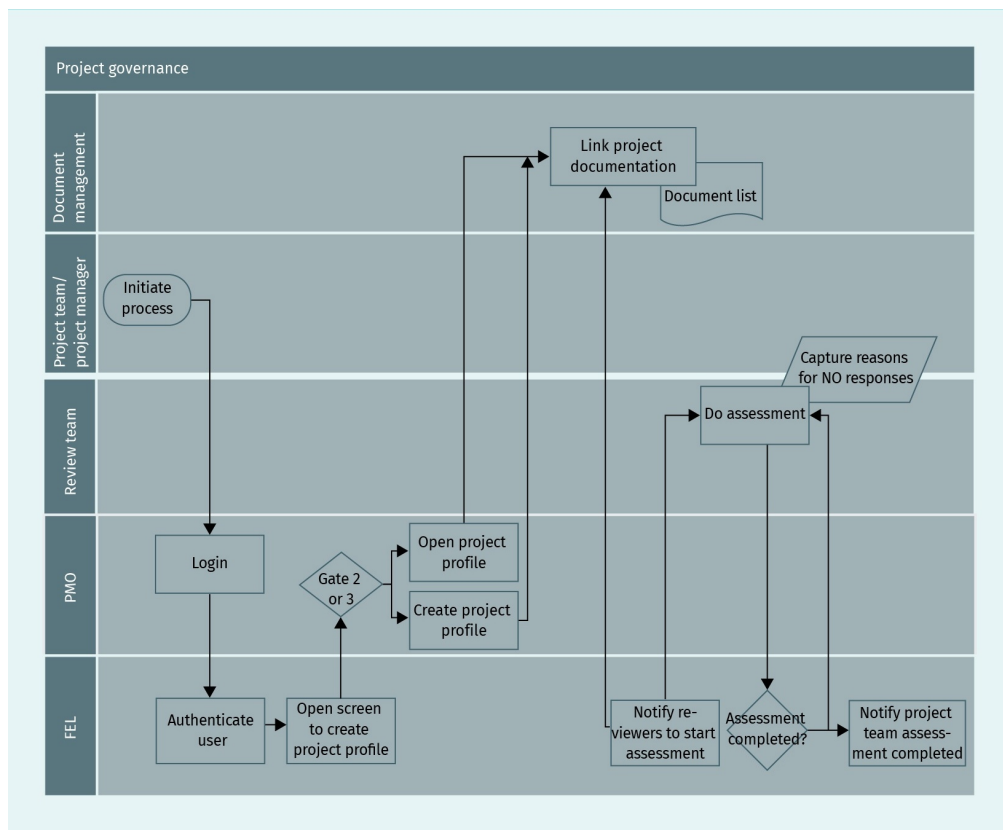
Figure 32: BPMN Elements



Source: Venter, 2021.

An example of a BPMN diagram is shown below. The diagram illustrates the flow of activities in the “Project governance” business process, which describes the process of assessing a project’s front-end loading (FEL) status, involving stakeholders and representatives from the Project Management Office (PMO), a review team, the project team and project management, and document management.

Figure 33: Example of a BPMN Diagram



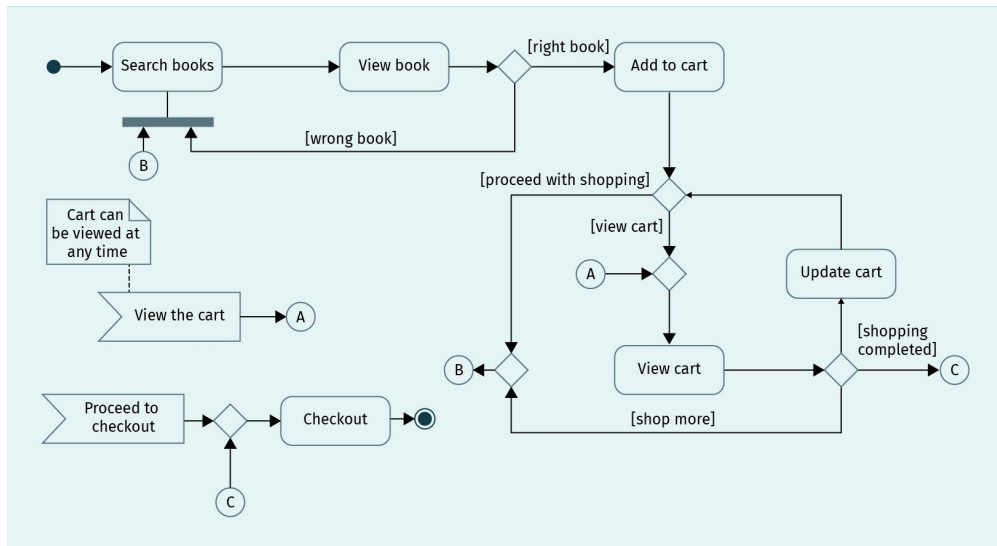
Source: Venter, 2021.

The use of BPMN does not eradicate the need for other system and software-centric notations to visualize integration and detail. For example, notations that continue to be applied in practice are Event-Driven Process Chains (EPC) and Petri nets. However, EPC lacks the standardization typical of notations such as BPMN and UML. Modeling using Petri nets are less expressive and can become quite complex relatively quickly; they may therefore be regarded as less suited to collaborating and communicating with non-technical users and business representatives (Kossak et al., 2014). BPMN diagrams and UML activity diagrams are notably similar, and activity diagrams effectively model business processes. UML activity diagrams have been redesigned in current versions of UML (2.0 and later) in terms of syntax and semantics, to enhance the capability of these diagrams to represent business processes (Geambasu, 2012). The symbols used in BPMN and activity diagrams are similar, with the exception that UML activity diagrams sometimes use groups of symbols to represent elements, whereas BPMN uses a single symbol. This is because BPMN uses complex symbols to holistically describe information (Geambasu, 2012).

An activity diagram can also be used to view detailed actions taken by, e.g., a customer that uses an online shop to purchase books. In the following figure, partitions are not used, and it is assumed that checkout includes both registration of a new user and log-in. The circled letters indicate where additional information is provided, or where additional

information can be added later when more detail is included or the work flow expands to include more activities. For now, using letters as indicators keeps the diagram simple and uncluttered.

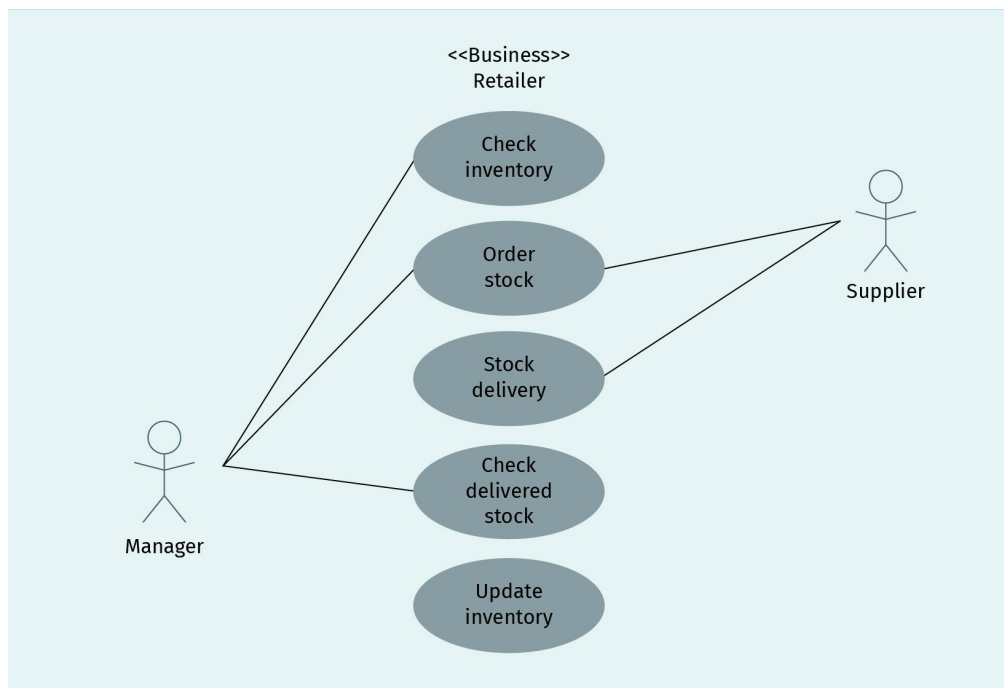
Figure 34: Example of an Activity Diagram Indicating Detailed Actions



Source: Venter, 2021.

Furthermore, use case diagrams can also be used to model business. Use case diagrams describe the functionality of a system from the user’s perspective. They show an overview of a business process in terms of requirements for a system, not in terms of procedures. The example of a use case diagram seen previously indicates, e.g., that a system is required to facilitate the purchasing of equipment. However, use case diagrams can also be used to focus on the business function, process, or activity. Rational Unified Process (RUP) introduced business use cases. RUP is “an ‘architecture-centric’ process” (Avison & Fitzgerald, 2006, p. 462) that applies three key concepts: use cases, architecture, and iteration. The UML use case elements of actor and use case will then be extended to include the business actors and business uses, and it will describe the business boundary rather than the subject (the system boundary). Business use cases are indicated with an ellipse containing a skewed line, as shown in the figure below.

Figure 35: A Business Use Case



Source: Venter, 2021.

SUMMARY

The interconnected nature of software processes means that various factors, people, and circumstances affect them, making them difficult to describe and manage. Software process definition and modeling aim to simplify and optimize the design, development, and implementation of software-intensive systems. They also help to optimally manage risk, software verification, and maintenance.

Standardized modeling notations, based on meta-models, facilitate operational visualization of systems. UML is a standardized object-oriented modeling language that is widely applied in software engineering. SysML is also applied in software engineering; it facilitates the modeling of systems-of-systems. Meta-models, such as MOF, describe the notation applied in meta-models.

Software systems consist of one dimension of organizational landscapes. In order to have a more comprehensive view, interfaces between business processes are modeled and detailed level notations are applied. Notations such as MPV-L and BPMN are used for this.

UNIT 3

BASIC SOFTWARE PRODUCT LIFE CYCLE MODELS

STUDY GOALS

On completion of this unit, you will have learned ...

- the strengths and weakness of waterfall models.
- about the need for verification and validation in software development.
- how models, such as the Rational Unified Process (RUP), support the entire software life cycle process.
- techniques that can be used to iteratively and incrementally evolve software.

3. BASIC SOFTWARE PRODUCT LIFE CYCLE MODELS

Introduction

Software is planned, designed, developed, and maintained using phased methodologies that encompass “procedures, techniques, tools, and documentation” in order to effectively “plan, manage, control, and evaluate” these projects (Avison & Fitzgerald, 2006, p. 24). Different approaches are typically based on different philosophical views. Accordingly, methodological approaches are applied according to a model and in the context of a framework. Over time, various models and frameworks have emerged, which are suitable for specific purposes and have specific areas of application. Since the software or system development life cycle (SDLC) emerged in the early 1960s, current models and methodologies continue to incorporate its elements.

Waterfall model
A predictive, prescriptive, and plan-driven software engineering methodology for the SDLC.

The **waterfall model** is often referred to as the first and most traditional development approach. According to Avison and Fitzgerald (2006), the waterfall model emerged in the “early-methodology area”, which was “characterized by an approach [...] focused on the identification of phases and stages [...] thought to help control and enable better management [...] and bring a discipline to bear” (p. 577). This model is still widely used and has its advantages, but it also poses significant challenges and risks. As a result, variations of this model that attempt to counteract these challenges have emerged. The waterfall model with feedback, the sashimi model, and the V-model are examples of variants of the traditional waterfall model. The waterfall model with feedback and the sashimi model enable iterations back to previous phases, whereas the V-model focuses explicitly on verification and validation during each phase in order to develop high quality products.

With the emergence of object-oriented analysis and design, component or matrix-based models, such as the Rational Unified Process (RUP), were introduced. RUP supports the entire software development life cycle; it iterates work within phases until goals have been met. The RUP philosophy supports principles of Agile as it facilitates iterative development, but is still plan-driven and architecture-centric.

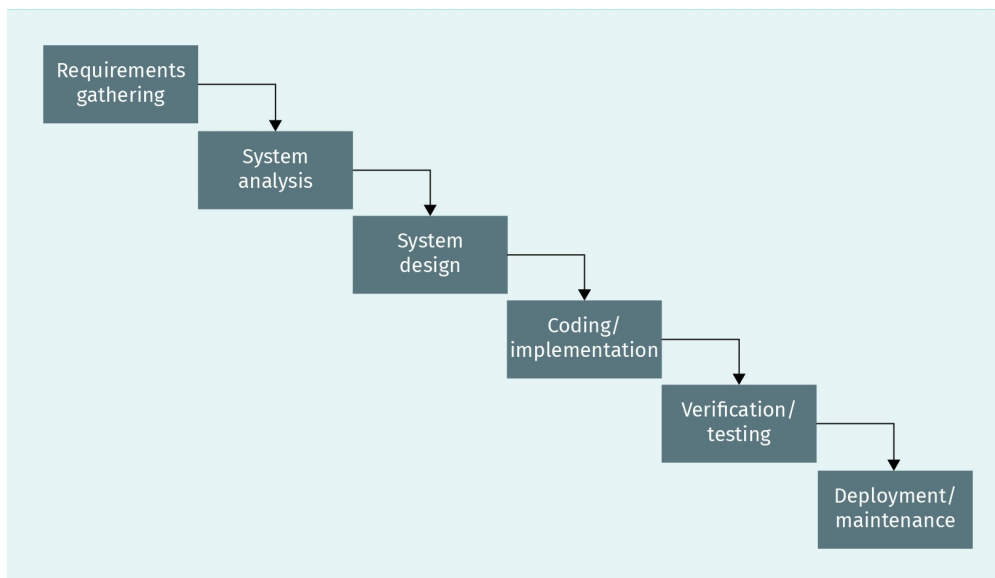
Iterative, incremental, and evolutionary development approaches emerged to better manage risks and enable better responsiveness to user’s requirements during development. Accordingly, prototyping is applied to derive and refine requirements, or to evolve solutions as requirements arise and develop. Boehm’s spiral method also presents an incremental and risk-driven approach; it involves basic phases to set objectives, assess and reduce risks, develop and validate, and plan ahead. However, it is non-prescriptive, as it enables the incorporation of any method or combination of methods.

3.1 Waterfall Models

The sequential waterfall model is the oldest and most widely known software engineering and IT project management methodology for the software or system development life cycle (SDLC) and is still used to develop software systems today (Shukla & Saxena, 2013). Designers and developers aim to predict what will be required in advance and then plan and work accordingly. The phase-by-phase and chronological progress of this SDLC flows downward, similar to the flow of a waterfall.

According to this model, a software development project typically starts with the requirements gathering phase, continues to the analysis and design phases, moves to development and verification, and concludes with the deployment and maintenance phase, where the solution is released to and used by the customer. A new phase can only begin upon completion of the previous phase. The waterfall model's phases and sequential nature are illustrated below.

Figure 36: The Waterfall Model



Source: Jamsa & Harkiolakis, 2019.

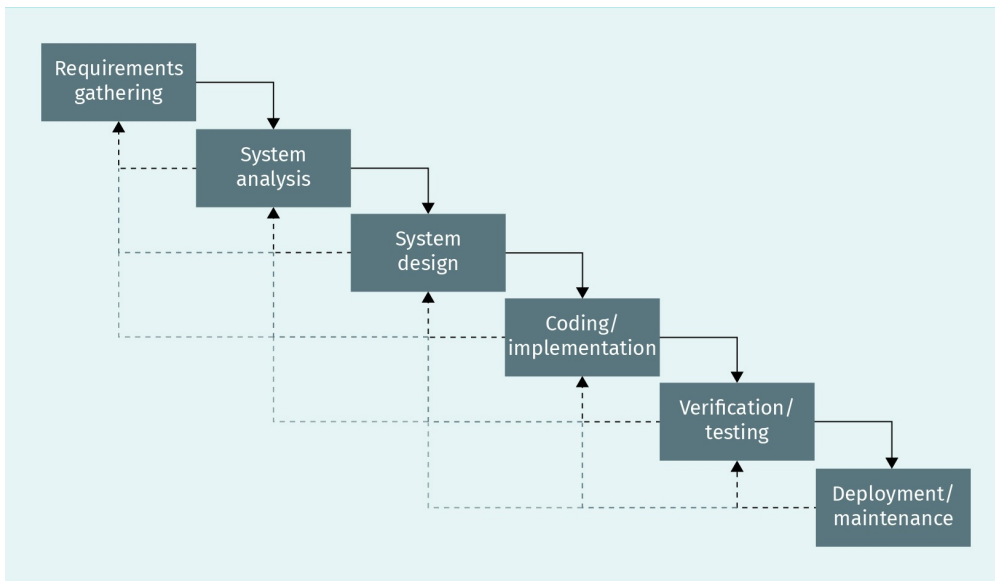
In terms of scheduling, roughly 20 to 40 percent of time is spent on the first three phases: eliciting requirements, analyzing, and designing the system according to these requirements. 30 to 40 percent of time is spent coding and developing, and the remaining time is used for testing and deployment activities. The benefits of predictive modeling can only be realized when developers thoroughly understand all requirements from the outset. They are then able to analyze and design the software system accordingly and prior to the development phase. Hence, it is recommended that sufficient time is allocated for these initial phases.

The waterfall model has a number of strengths. Standardized activities are defined in detail, and detailed and high-quality documentation is generated concurrently with other activities (Matković & Tumbas, 2010). The sequential application of these stages facilitates the phased development of a stable solution and is ideal for small and short-term projects where requirements are not likely to change during the project life cycle and for projects where quality is of critical importance. If all requirements have been properly considered in advance, the waterfall model enables improved planning and structuring of phases. Use of standardized, documented practices enables the drafting of complete specifications. It also enhances communication practices within teams and with stakeholders. In addition, project progress can be easily monitored and controlled, and is clearly visible to all stakeholders (Avison & Fitzgerald, 2006).

This model also presents notable challenges. It is difficult to accommodate changes, a consideration that may be relevant in cases where requirements can only be tested and verified during development or close to the end of the project; it does not allow stages to overlap (Hijazi et al., 2012). Iteration between stages is possible, but costly, so the strict sequential nature with which this model is typically applied results in an inability to iterate back to previous phases. This is challenging in cases where the requirements are not perfectly clear at the beginning of the project. The waterfall model also lacks feedback between phases; it presumes that all requirements are known and have been accurately gathered at the beginning of a project (Kaur & Sengupta, 2011; Prakash et al., 2012). It is often associated with high development costs. Faults that were not removed in earlier phases will have a negative impact later on in the process (Matković & Tumbas, 2010). Quality can only be ensured through proper planning and execution of each phase; all aspects of a phase must be addressed before moving to the next, but this can be difficult to execute in reality.

Variations of the waterfall model that attempt to overcome these challenges have emerged. Variations of the traditional waterfall model have one feature in common: they all add iterative elements. A waterfall model that allows feedback and iteration to previous phases is an example of such a variation. Another variation is the sashimi model, which allows phases to overlap. However, with these variations, iterations to previous phases and overlapping of phases must be carefully planned in order to limit the negative effect that they will have on cost and schedule. The further back an iteration goes, and the deeper the levels of overlap, the bigger the impact on both cost and schedule. The waterfall model with feedback is illustrated below.

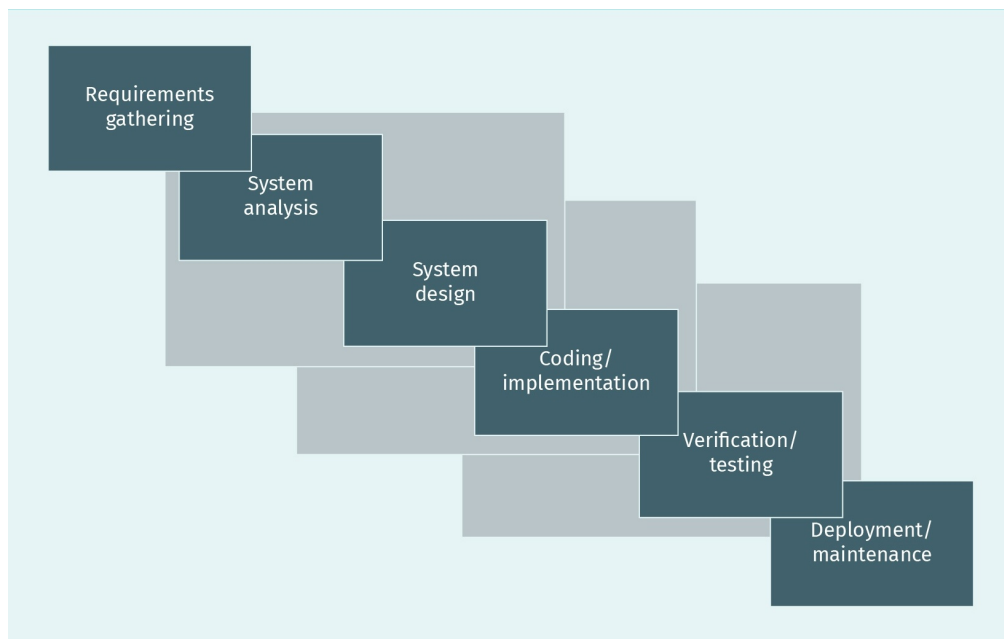
Figure 37: The Waterfall Model with Feedback



Source: Jamsa & Harkiolakis, 2019.

The sashimi model is a variant of the waterfall model that allows phases to overlap. It is also referred to as the sashimi waterfall or the waterfall with overlapping phases. The name of this model was derived from the resemblance that it has with the Japanese dish, sashimi, which consists of overlapping thin slices of fish. For example, during the first phase, a portion of the requirements will be defined, allowing team members to proceed with analysis and design. Similarly, at a later phase, when a portion of the coding work has been completed, it will be tested while other sections of the system are being developed or designed. Greater overlaps can also be considered to allow different parts of the project to move forward at different paces, as long as it does not cause conflict or is dependent upon an unfinished portion. For example, the solution can only be deployed when all parts have been tested. In the sashimi model, the documentation is one unified document, unlike the traditional waterfall model, which prescribes the documenting of each phase separately. The volume of documentation is thus significantly reduced (Matković & Tumbas, 2010). The sashimi model is shown below.

Figure 38: The Sashimi Model



Source: Jamsa & Harkiolakis, 2019.

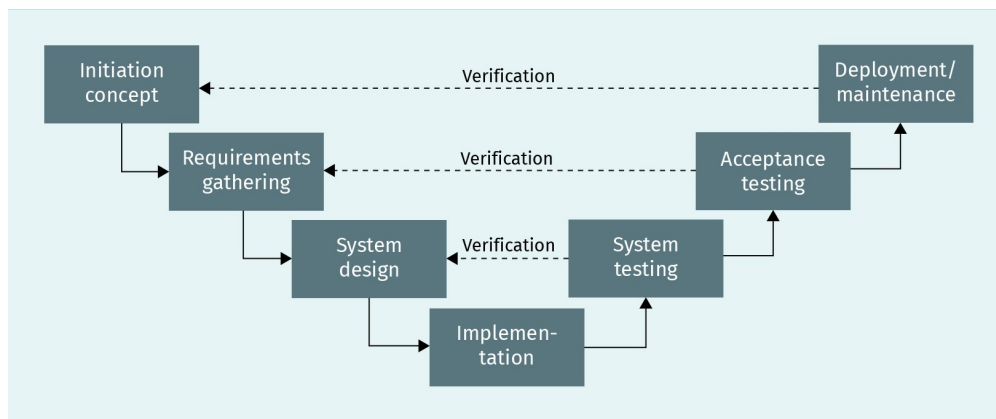
The sashimi model has the advantage of accelerating development, and the overlaps enable optimal use of resources and expertise across various phases. For example, a database designer may start to develop the database tables and indices as soon as the primary requirements have been identified and before the finalization of user interfaces. Similarly, a network designer may begin setting up hardware, such as routers and switches, prior to finalization of the network topology, which also enables the exploration of specific aspects of a solution, and facilitates increased understanding before final decisions are made. Valuable insights are gained and risks can be reduced. In this way, the sashimi model facilitates receiving feedback through an internal loop process and shapes the direction of the project from later phases back to earlier ones. This model enables, for example, the identification of errors made during design while design is still ongoing; however, it also presents some challenges. For example, key development milestones are unclear, monitoring individual activities can be difficult, and communication can be hindered (Matković & Tumbas, 2010).

3.2 The V-Model

The importance of early and timely **verification and validation (V&V)** was explored by Boehm (1984). In contrast to the relatively sequential nature of waterfall models, Boehm highlighted the need to continuously verify and validate software requirements and design specifications in order to reduce costs, improve reliability and maintainability, and increase a software's user-responsiveness. Accordingly, the V-Model (also known as the validation and verification model) is a popular alternative to the waterfall model. It is applied **to develop** relatively small and technical software systems that require high levels

of safety and security. The V-Model is a waterfall, but one that forms a V shape. The left-hand side follows a deductive approach, decomposing tasks into more detailed ones, while the right-hand side follows an inductive approach towards higher levels of abstraction where the various portions of the project are integrated. The model is illustrated below.

Figure 39: The V-Model



Source: Jamsa & Harkiolakis, 2019.

In this model, a product is essentially developed in a step-by-step fashion in the left branch, while results are integrated and verified in the right branch. Continual verification is emphasized; each constructive step (level of construction) on the left-hand side is supported by a verification step (level of abstraction) on the right-hand side. At the lowest level, testing aims to verify that code runs and executes correctly. Upon further progression, verification extends to confirming whether requirements have been met, and ultimately, whether customers are satisfied with the final product.

3.3 Component or Matrix-Based Models

Component or matrix-based models, such as the Rational Unified Process (RUP), were introduced in the 1990s to describe object-oriented development. RUP is best described by Avison and Fitzgerald (2006) as a “processified software engineering process” and software development practice that focuses on “software architecture and iterative development” (p. 461). It was jointly developed by the companies Rational Software Factory and Objectory, which were later purchased by IBM. The process incorporates Unified Modeling Language (UML) principles and concepts as it should be used in conjunction with UML. The objective of RUP was to have a process to support the complete software development life cycle (Jacobson et al., 1999). RUP uses UML use case and includes object-oriented analysis and design methods. The RUP philosophy encompasses some Agile principles, but it is a plan-driven approach that contains many guidelines.

The design process is centralized around architecture views. At first, the architecture is defined at a high level; it then evolves as the requirements for the software system evolve. This model is similar to the risk-based spiral model (Boehm's spiral method) in that it facilitates incremental and iterative evolution of requirements while simultaneously mitigating risks. RUP's aims are to reduce the size and complexity of products that are to be developed, to streamline the development process, to create teams that are more adept and proficient, and to exploit automation through the use of integrated tools (Boehm & Turner, 2004). There are two versions of RUP (RUP Classic and RUP for Small Projects); however, RUP is typically applied to large projects, as it remains relatively difficult to tailor to smaller projects.

RUP differentiates between content of work to be done (indicated by workflows) and time to complete this work (indicated by four phases). A workflow includes the sequenced activities that produce a valuable result, and RUP includes workflows for principle (engineering) work as well as auxiliary (support) work. The principle workflows are business modeling, requirements, analysis and design, implementation, testing, and deployment. The auxiliary workflows are configuration management, change management, project management, and the environment. Workflows are distributed (phased) over time. The phases are as follows: inception, elaboration, construction, and transition. Each phase is iterated until all pre-defined, phase-specific criteria have been met, and when a suitable and detailed plan for the next phase has been developed. RUP also applies the concept of a worker—it does not refer to an individual, but a role that is performed in the development process and within the workflows. A worker performs a set of related activities that aim to manipulate an artifact. Artifacts include, for example, use cases, models, documents, plans, and products.

3.4 Iterative, Incremental, and Evolutionary Development

Requirements for software are often unclear and hence change during development. In some cases, requirements may only surface fully after deployment. The uncertainty and changeability of software requirements led to the emergence of iterative, incremental, and evolutionary development approaches. These entail repetitions of actions in order to quickly obtain feedback, and therefore reduce risk. Prototyping and Boehm's risk-driven spiral model are some examples.

Prototyping is not a new concept. Royce (1970) argued that, even when following a typical waterfall approach, a **prototype** should be used in the first iteration in order to derive **apt specifications**. A subsequent iteration is then used to apply learnings and provide a final product. Prototyping can also be used evolutionarily to develop a solution, i.e., to accommodate changing and evolving requirements during development. An example of this is when a prototype is developed, evaluated, and reworked until all stakeholders are satisfied (Gull et al., 2009).

Prototype

“The entire process done in miniature, to a time scale that is relatively small with respect to the overall effort” (Royce, 1970, p. 334)

A prototype is a learning tool; it assists in reducing uncertainty and facilitating learning about, e.g., a system to be developed. A prototype can be used to verify the user interface or confirm whether a technical construct works as intended. Prototyping facilitates rapid development and stimulates creativity. It can also reduce cost, enhance quality, and increase user involvement (Matković & Tumbas, 2010).

Floyd (2011) suggests the following steps to effectively use prototyping as a learning tool:

- functional selection. A prototype will have limited functionality, so it is vital to define what to include (and exclude), based on the prototype's purpose.
- construction. A prototype must be built according to its definition.
- evaluation. Learn from the prototype by applying it for its purpose.
- further use. Depending on its reason for being developed, the prototype can be evolved further or discarded.

There are different kinds of prototypes. Floyd (2011) distinguishes between them as follows:

- Exploratory prototypes are useful to identify and refine requirements, as well as to develop a shared understanding of expected features among developers and stakeholders.
- Experimental prototypes are used to experiment with a solution in order to validate it, e.g., to confirm a functionality or a solution's technical feasibility. Both exploratory and experimental prototypes can either be developed further or discarded.
- Evolutionary prototypes are revised and progressively extended until they evolve into a final product. During this evolution, the requirements, specifications, and design evolve until the customer is satisfied. The end result is working software.
- A throw-away prototype is rapidly developed with the aim to discard it once it has served its purpose. It is typically used to derive and develop requirements and specifications in a relatively short space of time, validate requirements, and detect and resolve issues early in the process.

While evolutionary development has various advantages and often results in increased user satisfaction, there are also some risks to consider. It can be successfully applied to larger and more complex systems if an initial and low-functionality version of the final product can be evaluated early on in the process (MacCormack, 2001). However, one of the shortcomings of evolutionary prototyping is that expectations must be astutely managed to ensure that customers do not confuse an incomplete prototype with a final product. Constant changes to the product may also result in poorly structured software that is difficult to maintain, the process followed may not be sufficiently visible, and progress may be difficult to manage (Sommerville, 2011). It can also be difficult to accurately assess resource requirements up front and plan for integration, and documentation tends to remain incomplete (Matković & Tumbas, 2010).

Boehm introduced the spiral method in 1988. It is a risk-driven (rather than document or code-driven) approach that can incorporate any method or combination of methods (Boehm, 1988). It endeavors to integrate evolutionary and specification-based develop-

ment approaches (Sommerville, 1996). The spiral model contributed to the promotion of iteration in software development. It “was devised as an ode to iteration” (Matković & Tumbas, 2010, p. 168).

Boehm (1988) illustrates this model in the form of a spiral with loops. Each loop represents a non-prescribed phase of a software process. It is split into the following four sectors: objective setting, risk assessment and reduction, development and validation, and planning. This model does not iterate implementations; it revisits each phase, until final implementation, to manage and reduce risks (Avison & Fitzgerald, 2006). As the team continues to revisit phases, their understanding of the solution evolves, risks are identified and mitigated, and a solution can evolve or be refined by means of incremental development of the solution or prototyping, for example. Activities are initially highly abstract, progressing gradually to become more detailed.

The spiral model shares some of its philosophical footings with Agile development. For example, it has the following advantages:

- Functional software can be produced relatively quickly.
- Various approaches can be combined.
- Continuous risk-assessment results in appropriate responsiveness.

However, it remains a systematic and plan-driven approach and, as such, retains the advantages that come with following a well-planned method. One of this model’s main drawbacks is that proper risk analysis requires specific expertise, which can be costly and sometimes not readily available for smaller-scale projects (Matković & Tumbas, 2010).

SUMMARY

Predictive and plan-driven models are useful for small projects and quality-driven projects. They facilitate effective (strict) management of small software engineering projects and are easy to plan, document, follow, and monitor. The strictly sequential nature of the traditional waterfall model makes it rigid and unable to accommodate new or changing requirements. To counteract this disadvantage, variations, such as the waterfall model with feedback and the sashimi model, were introduced. Another variation of the waterfall model, the V-Model, emphasizes continuous verification and validation. It is also suited for projects driven by reliability, correctness, and quality. It is suitable for projects that demand high levels of safety and security, or projects in a financial environment.

Component or matrix-based models, such as RUP, emerged with object-oriented development. RUP aims to support the complete software development life cycle. It embraces Agile principles, but remains plan-driven and prescriptive. However, unclear and changing requirements have led to the emergence of iterative, incremental, and evolutionary

development approaches, such as prototyping and Boehm's risk-driven spiral model. They involve coupling repetitive actions with constant feedback in order to increase responsiveness and reduce risk.

UNIT 4

AGILE AND LEAN DEVELOPMENT PROCESSES

STUDY GOALS

On completion of this unit, you will have learned ...

- about the characteristics of Agile development and common Agile practices.
- the specifics of the Scrum method.
- about Kanban and lean development.
- how to scale Agile methods using the Scrum of Scrums (SoS), Large-Scale Scrum (LeSS), and Scaled Agile Framework (SAFe).
- about the practicality of applying hybrid processes.

4. AGILE AND LEAN DEVELOPMENT PROCESSES

Introduction

Agile development methods emerged to counteract the challenges of waterfall development approaches. The Agile manifesto makes it clear that people, working software, collaboration, and responsiveness must be prioritized; technical tools only support the people that will develop and use the software. Agile development is iterative and incremental; it applies short increments in order to develop solutions that meet customer expectations.

Scrum, as an Agile development method, is widely used and applied successfully to develop smaller-scale software systems. Scrum is not strictly a software development method, but it guides teams to successfully develop useful artifacts. It incorporates a number of guiding principles and values, team roles, specific events, and artifacts. Common practices are applied to make Agile development successful. These include the product backlog, sprint backlog, increment, task board, and burn down chart.

The advantages and successes of Agile approaches, such as Scrum, made them attractive, and attempts to scale them for larger and more complex projects soon followed. Scaled Agile approaches, such as the Scrum of Scrums (SoS), Large-Scale Scrum (LeSS), and the Scaled Agile Framework (SAFe), are progressively refined and applied. These are used as overarching organizational frameworks, rather than project management or software development approaches, to manage and guide the planning, development, and integration of software systems into organizational structures.

4.1 The Agile Manifesto

In the 1970s and 1980s, business software was primarily developed using waterfall approaches, i.e., well-structured and sequenced processes that consist of distinct phases that are methodically executed. Activities, to be completed prior to moving to the next activity, include project initiation, requirements discovery and gathering, solution architecture and design, development, testing, and deployment. Despite their advantages, sequential waterfall development processes also have significant shortcomings, for example, the inability to incorporate requirements that are not fully known at the beginning of a project. Since they do not allow for iterative feedback from users, software systems can also be carried fully to completion and deployment, only for issues to then be discovered. For example, the interface was not sufficiently intuitive, more data should have been captured, the processing logic was incorrect, or the output did not serve the required purpose. These issues result in an expensive and time-consuming redesign process, which involves software changes and retesting. In addition to delays to project delivery and additional costs, it can also result in dissatisfied business users. Therefore, Agile (light) devel-

opment was introduced in the late 1990s as an alternative to traditional (heavy) methods (Highsmith, 2001). These methods promised to be more responsive, following a more iterative and incremental approach.

Waterfall models established strict rules for working, as well as documentation within the development process. Proper and complete documentation of activities, and the software system, are only useful if they work well and meets the customer's requirements. However, it is also true that the continuous application of rigid processes and the drafting of documentation ties up many resources that are then unable to react to changes or continue with development work. In response to this challenge, seventeen leaders in the software industry created and signed the Agile Manifesto in 2001 (Beck et al., 2001a). The Manifesto supports customer-centric and streamlined ways of working; it does not disregard processes and tools, but favors customer-centricity and responsiveness. The values of the Agile Manifesto are as follows (Beck et al., 2001a):

- Individuals and interactions over processes and tools,
- Working software over comprehensive documentation,
- Customer collaboration over contract negotiation, and
- Responding to change over following a plan (para. 2).

Beck et al. (2001a) explicitly state that, with these principles, customer-centric aspects such as individuals and interactions, working software, customer collaboration, and response to change, are prioritized over technical aspects, such as processes and tools, comprehensive documentation, contract negotiation, and following a plan.

The Agile Principles prescribe the following (Beck et al., 2001b):

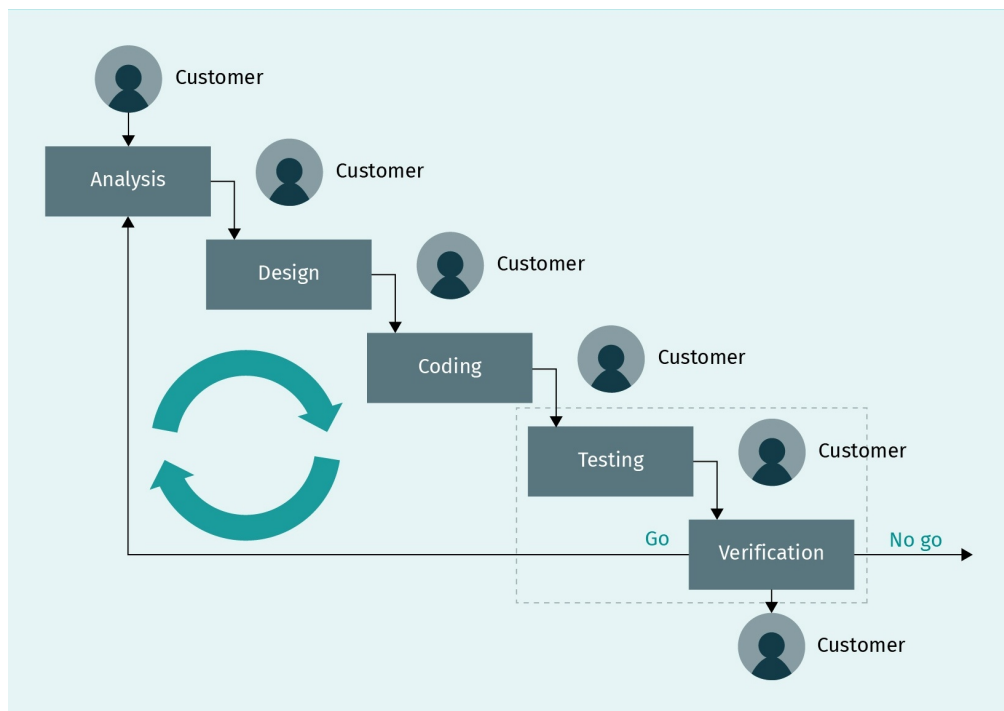
- satisfying customers with valuable software that is delivered early and continuously,
- harnessing change throughout the process,
- using shorter time scales,
- continuous collaboration between business users and developers,
- building projects around individuals that are supported and hence motivated,
- favoring face-to-face conversations to convey information,
- measuring progress based on whether software works,
- following a constant and sustainable development pace,
- focusing on technical excellence and good design,
- maintaining simplicity,
- self-organizing teams,
- regularly reflecting within the team about how to work more effectively, and
- adjusting behaviors as required.

The Agile Process

In practice, business analysts, architects, designers, and developers continuously collaborate with business users to ensure that business goals and user requirements are understood and met. Accordingly, Agile teams follow an iterative and incremental approach to development. The goal of each iteration is to produce a working and verifiable solution with the customer involved in every step. Working versions are referred to as revisions.

The solution is revised (improved) with each successive iteration. At the end of an iteration, a final revision is tested and debugged, producing a build. Completed functions or requirements are released as minor releases and the addition of significant new features are major releases. Releases follow a strict versioning scheme in the format “major.minor.build.revision”, e.g., version 4.2.5.9 indicates that it is the fourth major release, the second minor release, the fifth build, and the ninth revision. Iterations are relatively short, ranging from a week to a couple of months depending on the project. Each iteration is, fundamentally, a waterfall structure in itself; it typically includes all of the development phases, i.e., analysis, design, coding, testing, and verification (Coram & Bohner, 2005). This is illustrated below.

Figure 40: The Agile Process



Source: Jamsa & Harkiolakis, 2019.

Sprints
A series of time boxed incremental iterations. The duration of each sprint typically does not exceed one month.

Cadence
This is an Agile term for the number of days or weeks in a sprint or release.

Agile projects are divided into short phases (**sprints**) to build working software (increments), enabling prompt feedback, which is used to adjust and improve as required. Daily meetings (Scrums) are 15-minute check-in sessions where team members share what they have worked on the previous day, what they will work on the present day, and whether they are experiencing or expect any difficulties. Sprints are time-boxed to manage scope, develop a **cadence**, and facilitate release planning. Teams should be able to deliver working software at the end of each sprint.

Challenges involved with a pure Agile approach include the following:

- distributed teams. Agile is designed for face-to-face collaboration, for example, for Scrum meetings and requirements gatherings.

- resistance to change. If previous approaches have been followed, teams may not be willing to adapt to a new Agile approach.
- less predictability. The iterative and incremental approach to requirements discovery leads to a continuous need to adapt to changes.
- increased demands on developers and subject-matter experts. This is due to the iterative and incremental discovery and testing of requirements.

Examples of Agile approaches include Extreme Programming (XP), Dynamic Systems Development Methodology, Feature-Driven Development, Crystal Methodologies, Adaptive Software Development, and Scrum. Of these approaches, the Scrum method was the first to be formalized and published (Sutherland, 2001).

4.2 Scrum

Scrum is an Agile approach where a team works together to advance development in short time spans. The “Scrum” idea was initially conceived as a metaphor that described the need to apply six interrelated characteristics in a holistic manner in order to innovatively develop new products. According to Takeuchi and Nonaka (1986), to “move the Scrum downfield”, the following characteristics were identified to be utilized for effective product development:

- built-in instability,
- self-organizing project teams,
- overlapping development phases,
- organizational transfer of learning,
- subtle control, and
- multilearning (p. 138).

These characteristics, when applied holistically, result in a new set of dynamics. This practice, named Scrum (short for Scrummage, the rugby term that describes how two teams form a circle and use their feet to gain possession of the ball after it is thrown into the middle of the circle), was later applied in the field of software engineering. It is characterized by the consistent organization of activities in short cycles and self-organizing teams.

Ken Schwaber and Jeff Sutherland co-presented Scrum in 1995. Scrum is documented and defined in *The Scrum Guide* (Schwaber & Sutherland, 2020). It does not define any specific software engineering-related roles or activities. Instead, it comprises a set of values, a team, events, and artifacts. Guidelines and rules prescribe how these fit together. Scrum applies fixed time and cost to control requirements rather than fixed requirements that control cost and schedule, as is often the case with traditional project management approaches. Fixed time and cost are achieved by means of time boxes, collaborative ceremonies, a prioritized product backlog, and frequent feedback cycles. Scrum is a framework used to plan and structure work; it is not a prescriptive methodology. As an Agile approach, it provides structure for delivery while simultaneously leaving specific practices to be followed, which are for the team to determine.

Scrum follows an empirical process and is based on the epistemological assumption that knowledge is created through experience. It builds a team's experience on a particular project using multiple iterations in order to optimize predictability and control risk. The development process is empirically controlled and supported by three pillars: transparency, inspection, and adaptation (Schwaber & Sutherland, 2020). Transparency ensures that the process and work are visible to those responsible for the outcome and the recipients thereof. This ensures a shared understanding of all visible aspects and a collective understanding of what it means when something is still ongoing or has been completed. Transparency also facilitates inspections of process artifacts to ensure compliance with the goals of the iteration. Through this, negative side effects and undesirable artifacts are made visible. Inspection facilitates adaptation. When inspections reveal deviations outside of the acceptable range, the team must adapt and adjust the process to (re)align outcomes with goals. The frequency of inspections must be determined by skilled inspectors in order to optimize, and not delay, the work. Scrum's five events provide cadence to aid inspection.

The Scrum Values

Scrum teams commit to embracing and following the values of commitment (to the goals of the team); courage (to do the right thing and overcome problems); focus (on the work and goals); openness (about the work and challenges); and respect (toward other team members as capable and independent professionals). These values direct the team's work, actions, and behavior (Schwaber & Sutherland, 2020).

The Scrum Team

"The Scrum Team is responsible for all product-related activities from stakeholder collaboration, verification, maintenance, operation, experimentation, research and development, and anything else that might be required" (Schwaber & Sutherland, 2020, p. 5). A Scrum team consists of one Product Owner, one Scrum Master, and a number of developers. The team is cross-functional and self-managing (i.e., self-organizing), and focus on the product goal. A team typically consists of a maximum of ten people working on one product. Multiple cohesive teams can focus on one (large) product, provided that the teams have a shared product goal, product backlog, and Product Owner. These roles are described by Schwaber and Sutherland (2020) as follows:

The Product Owner

The Product Owner is a person that acts as the proxy for the customer, user, or other stakeholders. The Product Owner has the authority to make decisions regarding the solution and holds the product backlog, i.e., a prioritized list of user requirements and features to be included in the solution. Product Owners

- are the liaison between the developers and stakeholders.
- gather and analyze requirements.
- communicate the vision to the team.
- define and prioritize items in the product backlog for each iteration.
- verify that the product meets the requirements.

- plan and announce releases.
- demonstrate the solution to stakeholders.
- keep all stakeholders informed regarding the project status.

They also create release plans, prioritize requirements, and approve results in each cycle. The Product Owner has contact with the team on a daily basis to answer questions and provide clarifications. They generally do not interfere with iterations when they are in progress, but can make changes to be incorporated in future iterations or even cancel future iterations.

The Scrum Master

The role of the Scrum Master is similar to that of a project manager, but should not be confused with that of a project manager in the traditional sense. A Scrum Master is a leader, but does not have authority to instruct the team. They are a resource to the team and do not impose on their self-organization. Instead, they create an ideal working environment for the team and shield them from external influences and disturbances. The Scrum Master removes obstacles, negotiates with those that are external to the team, ensures that the team follows good Scrum practices, inspires and challenges the team to improve, facilitates communication, and mediates discussions.

Developers

Developers are responsible for developing the solution. They create a plan, perform technical conceptualization, design, assure quality, and adapt as required. They manage themselves, deciding among themselves how the workload is distributed, developing relevant metrics and estimates, and reporting to each other during daily Scrum sessions. The team members hold each other accountable in order to work professionally. They form a part of an equal and egalitarian group, sharing duties and responsibilities.

The Scrum Events

Scrum events include: sprint, sprint planning, daily Scrum, sprint review, and sprint retrospective. Each event provides a formal opportunity to inspect work and adapt as required (Schwaber & Sutherland, 2020). A Scrum process begins with a customer who provides a clear vision and a set of product features, listed in order of importance. These features form the product backlog, which is maintained by the Product Owner as the customer's proxy. This initiates a series of time-boxed, incremental iterations, referred to as "sprints". Schwaber and Sutherland (2020) refer to sprints as "the heartbeat of Scrum, where ideas are turned into value" (p. 7).

A time box is a common Agile concept that describes an approach where the amount of time dedicated to an activity is fixed. If the planned work is not completed, the timeframe will not be extended; rather, what is ready will be delivered. In this way, tasks are concretely defined. A "sprint" is a container for a series of events where the durations are fixed; durations cannot be modified after the sprint starts. Typically, the duration of a

sprint does not exceed one month, as it is presumed that longer durations may lead to changes in requirements and definitions, resulting in increased complexity, risks, and even costs.

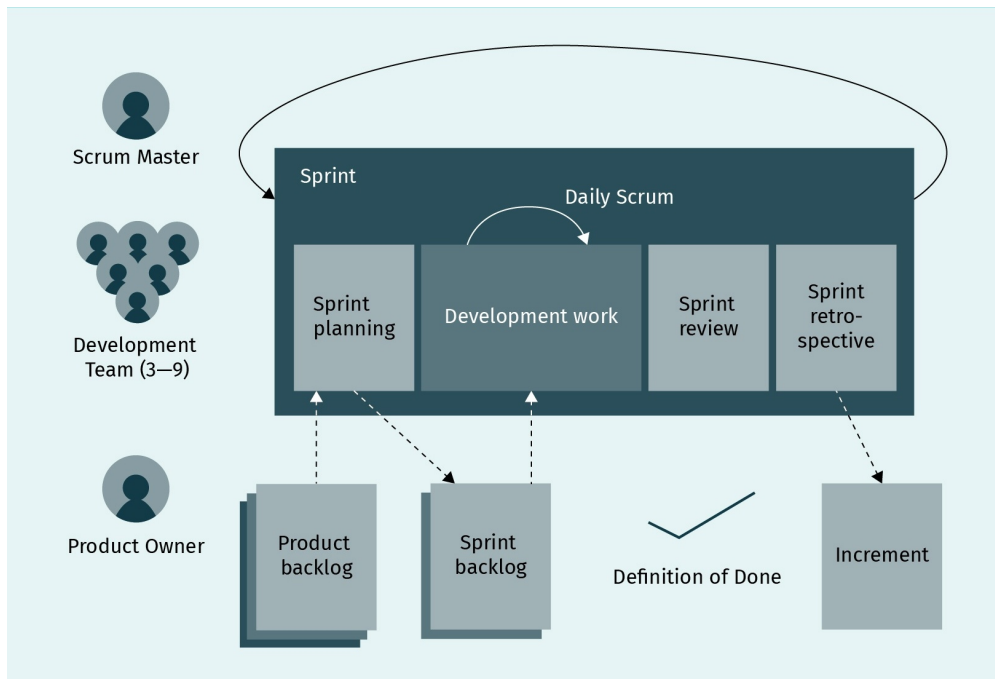
Sprint planning initiates the sprint—it results in a plan that addresses the following topics: “Why is this sprint valuable?” “What can be done during this sprint?” “How will the chosen work get done?” (Schwaber & Sutherland, 2020, p. 8). The team selects a list of items from the product backlog to be completed in the sprint and lists them in the sprint backlog. When everyone agrees, the work commences. No interruptions are allowed once work commences in order to ensure that the team can focus, meet the set goals, and complete the selected items.

Daily Scrums are 15-minute events to inspect progress and adapt the sprint backlog if needed. They lead to the sprint review and potentially to the sprint retrospective at the end of the sprint cycle. In Scrums, the team coordinates work and discusses and reviews progress. Additional tools, such as a task board and a burn down chart, are typically used in these sessions to aid the process. This is because many common tasks that must be performed are not defined as part of Scrum.

A sprint ends with a sprint review. This is an event, held over a maximum of four hours, where the team demonstrates a solution (an increment) to stakeholders and takes note of their feedback. The sprint retrospective follows this review, unless the developed increment has met all the specifications. It is time-boxed for a maximum of three hours. The sprint retrospective allows the team to reflect on the status of the project and consider improvements for the next sprint. They also consider ways to improve product quality, streamline work processes, and clarify what it means for a project to be “done”. The Scrum Master ensures that this meeting takes place and that it is conducted in a positive and productive spirit.

The sprint process continues to iterate until there are no further items in the product backlog. Then, a release iteration (release sprint) is planned in order to prepare for deployment. At this point, all relevant documentation produced during individual sprints are finalized, and any remaining defects are resolved. Further, physical items such as installation media, manuals, and packaging are prepared to be shipped. If required, system integration and testing take place. An acceptance review and the shipping of items to the customer signals the end of the project. An overview of a Scrum process is illustrated below.

Figure 41: The Scrum Process



Source: Kneuper, 2018, p. 104.

The Scrum Artifacts

There are three **Scrum artifacts**. These are the product backlog, sprint backlog, and increment. The product backlog commits to a product goal, the sprint backlog commits to a sprint goal, and an increment commits to the Definition of Done (DoD).

Scrum artifacts

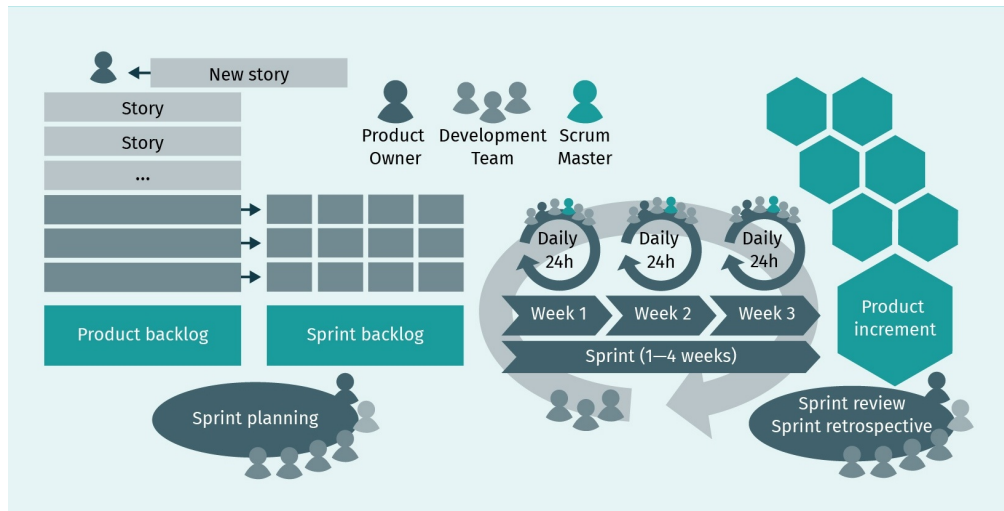
The three Scrum artifacts are the product backlog, the sprint backlog, and the increment.

The product backlog details the requirements, features, and their priorities; it can range from a draft of an initial idea to detailing fully-specified functionality. Initially, it may contain several rough outlines of goals and requirements. As the project progresses, requirements are refined, supplemented, and prioritized by the Product Owner. The product backlog contains the user stories that the team will be working on. A user story is a high-level definition of a requirement and is also used to estimate the relative time required to complete it.

The sprint backlog is a subset of the product backlog. It contains a list of the requirements that will be implemented during the sprint. The requirements included determine the batch size of the sprint, which depends upon the velocity of the team. The velocity of a team indicates how many (and which) of the requirements a team can complete during a cycle, and therefore how many (and which) of the requirements a team can load from the product backlog to the sprint backlog. It is derived from the quantity and scope of the functions, as was implemented in previous cycle(s). Velocity typically stabilizes after the first five to seven cycles.

An increment is a concrete and usable stepping stone; the sum of all increments equals the product goal. Work can only be considered an increment when it is completed and meets the DoD. An overview of the Scrum framework is illustrated below.

Figure 42: Scrum Framework



Source: Ismailovic, 2021.

Scrum Success Factors

The frequent sprints and the involvement of the whole team in the decision-making process are the core advantages of Scrum. Scrum success is a result of transparency and visibility, as well as empowerment coupled with accountability. The egalitarian nature of the team enables members to accommodate changes quickly and easily when required, resolve issues, reduce cost, and improve performance. The main challenge with Scrum is the individual and collective understanding that the team members have of the process. A more experienced team that is also familiar with the process will establish a working rhythm relatively quickly; less experienced team members, and specifically Scrum Masters, can ruin the development process. For this reason, a Scrum team will be assigned to their next project as an established team, rather than as individuals and to new teams. Scrum teams gain experience as they progress with a project; as they work through sprint backlogs, they gain knowledge from the partially delivered results and apply it to subsequent sprints.

The idea of self-organizing teams poses a challenge to traditional management and leadership practices, especially in relation to the management of teams, projects, and tasks. Self-organizing teams act independently; they have the authority and flexibility to choose and implement applicable methodologies as they see fit. They establish their own levels of authority, and may even choose their own team members. The aim of self-organization is empowerment. Team members are motivated to take initiative when choosing and completing tasks, take responsibility for their work, monitor their own progress, and seek support when issues arise. Self-organizing teams do not form instantly and spontaneously. This calls for inspiring leadership that guides and supports the team and encour-

ages team members to take initiative. It takes time for team members to gain momentum and form collaborative and self-sustaining hierarchies of responsibility and accountability within the team.

Communication is key for Agile development and Agile teams (Cockburn & Highsmith, 2001). Agile teams communicate frequently with each other and with customers (or their proxies) to ensure that the project remains on track, and to address and manage expectations. A solution must be inspected frequently to get feedback in order to identify and address any misalignment or issues and improve the solution. Frequent communication also ensures that Agile teams remain focused on quality and, hence, develop high-quality solutions.

4.3 Common Agile Practices

The most important management artifact in a Scrum process is the product backlog. Other Scrum artifacts that are also central for a successful project include the sprint backlog and the increment, as the result of the sprint. In addition, management tools that are frequently used in Scrum are the Definition of Ready (DoR), Definition of Done (DoD), task boards, and burn down charts.

The Product Backlog, Sprint Backlog, and Increment

During the sprint planning process, it is important that the Product Owner and team select a suitable set of prioritized and appropriately detailed items from the product backlog for the sprint backlog. As previously mentioned, the velocity of the team determines the number, as well as size, of elements in the sprint backlog. The sprint backlog should also only include as many elements as the team agree that they can reasonably manage. During the cycle, the number of elements in the sprint backlog is fixed—further elements cannot be added. Several increments can be created during one sprint, and work can only be considered as part of an increment when it adheres to the Definition of Done.

The Definition of Done

Readiness to release relates to the DoD. Agile teams strive to develop high-quality solutions. For this, they must write solid, high-quality code that will not require major modifications later on. Accordingly, teams emphasize proper coding principles, dedicate time to develop a common style, and ensure formal coding practices. A solution is only formally delivered once it adheres to the DoD, as indicated by a checklist. The sprint backlog elements that are worked on during a sprint are only marked as complete (done) when they have been completed according to the checklist and requirements have been met. Moreover, the sprint backlog entails a quality check for all the items on the list. Consistent application of these is also intended to ensure high process quality and must be enforced. Quality assurance and documentation are tedious tasks that are easily left behind, especially when working under the pressures of delivery or in high-stress situations. The DoD is continually updated and typically includes reference to specific items. The checklist specifies that an element adheres to the DoD when

- the design has been verified,
- all programming tasks (development) have been completed,
- all programming conventions have been observed,
- all the program code is available in the configuration management system,
- review of the code has been completed (if required),
- user documentation has been updated,
- all unit and module tests have been carried out successfully,
- the specified acceptance criteria have been met,
- acceptance tests with users have been carried out successfully,
- there are no malfunctions, and
- the solution is in production mode.

The Definition of Ready

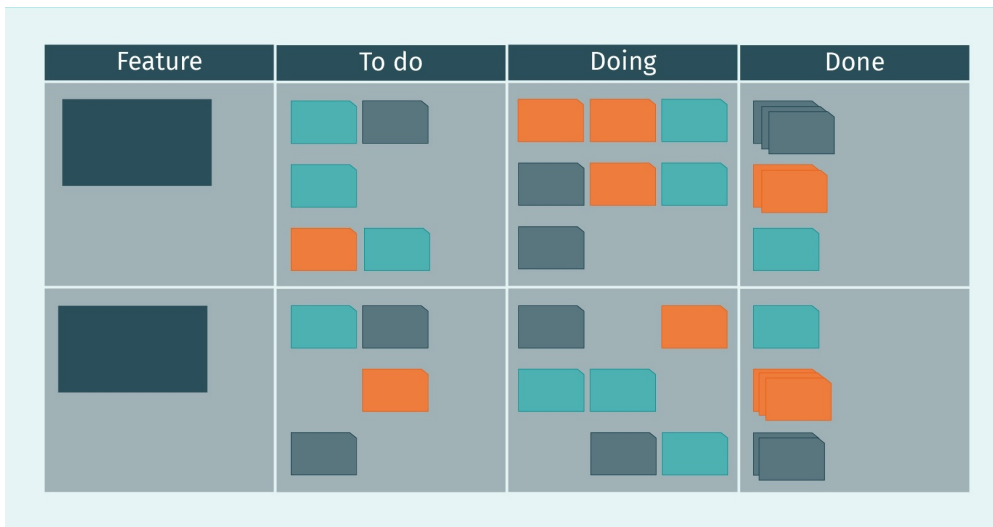
Requirements that are adequately defined for development and testing relates to the DoR. During sprint planning, the team negotiates the items to be selected for the next sprint and can only commit when requirements are sufficiently detailed, i.e., ready. The planning results in a checklist that indicates when items meet the DoR. The checklist specifies a clear definition in terms of development requirements, resulting business value post implementation, and pre-development enablers to be added. The checklist also specifies the following criteria:

- Rough estimating (sizing) indicates that it can fit within one sprint.
- No pending dependencies to external resources or elements will be needed during the sprint.
- In case of unavoidable live, external dependencies, adequate coordination has been arranged and will be closely tracked.

A Task Board

Task boards are often used during Scrum sessions. A task board, which is typically displayed in the common meeting area and populated with sticky notes, is used to track progress. A task board can indicate, for example, the work that is still pending (“to do”), the work that is still ongoing (“doing”), the work that has been completed (“done”), and any additional items, such as the user requirements and required features. An example task board is shown below.

Figure 43: A Task Board



Source: Jamsa & Harkiolakis, 2019.

Velocity

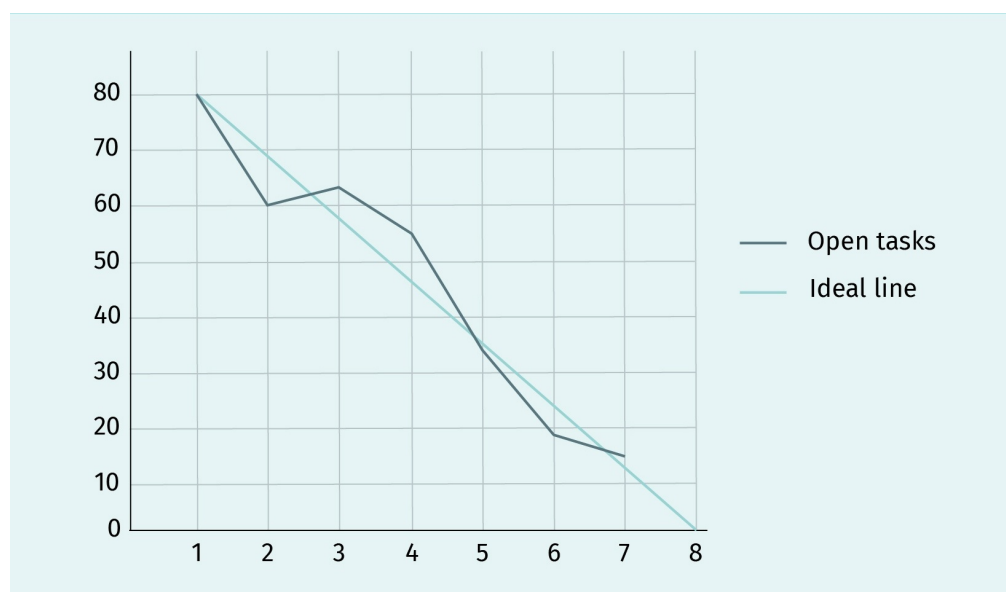
Velocity refers to the speed at which something moves in a certain direction. In Scrum, it indicates the amount of work that a team can successfully complete during a sprint. It is typically measured using the metric “story points” or “person days”. Story points are a relative and abstract measure used to put the size of different backlog elements into perspective. Higher points imply that it is more complex and that its implementation is estimated to be more costly. People are generally better at estimating in relative rather than absolute terms (Key, 2016), hence the frequent use of statements, such as “the element is approximately the same size” or “the element is significantly larger.” A standard number of achievable story points will typically emerge for a team once they have completed a number of sprints. Initial estimates will be imprecise, but they stabilize over time. Story points can be used to estimate time for tasks within a team, but cannot be transferred between teams. Story points do not indicate the actual scope of results that a team delivers. For example, a velocity of 40 story points for Team A can be very similar to a velocity of 35 story points for Team B, as it only indicates their own relative definition (El Deen Hamouda, 2014). Story points translate to stories, which are then implemented by the team. However, the number of person days is an absolute estimate; it indicates the time (in working days) that the team will need to complete a backlog event.

A Burn Down Chart

Burn down charts illustrate the progress of an Agile project. An Agile project’s critical path can change daily, making traditional means to track progress of items, e.g., a Gantt chart, inadequate. Instead, a burn down chart is used to show the current progress of the entire project, a work package, or a sprint. It is based on the finished elements of a product backlog or a sprint backlog, as the elements relate to time planned versus time still available. It indicates the velocity of development.

A burn down chart shows an ideal course (usually indicated by a linear line, i.e., the ideal line), the actual course (indicated by open tasks), and deviation from the ideal. If the actual course deviates to be above the ideal line, it indicates slower progress, and vice versa. At the end of the reporting period, the actual course should ideally reach the zero line, indicating that all open tasks have been completed, i.e., “burned”. The horizontal x-axis represents time; the first value is the starting point of a considered time period and the last point represents the end of the period. The vertical y-axis represents the number of the unit that is illustrated in the chart, e.g., the number of open tasks or the amount of resources (money or time) used. A burn down chart is illustrated below. It shows six intermediate measurement points between the start of the sprint, indicated by “1”, and the end of the sprint, indicated by “8”, as well as the default value of 80 open tasks.

Figure 44: A Burn Down Chart



Source: Brückmann, 2015, p. 59.

4.4 Kanban and Lean Development Processes

Lean manufacturing entails waste reduction, increased efficiency, and the elimination of bottlenecks. When considering lean practices in the context of software development, lean manufacturing also refers to increasing efficiency and elimination of bottlenecks. Reduction of waste is interpreted and applied in order to identify potential areas of improvement. Kanban and lean software development are defined vaguely. Nonetheless, the lack of prescribed practices is perceived as a strength. Lean manufacturing is non-prescriptive in the sense that it only introduces constraints for workflow visualization and to limit

work-in-progress (WIP) (Tanner & Dauane, 2017). In general, Kanban involves the application of general principles that, when applied by software development teams, assist them greatly in optimizing work in an Agile, iterative manner. These principles include the

- visualization of progress using a Kanban Board.
- limiting of WIP to maximize items flowing through.
- managing the workflow by focusing on completing prioritized items.
- making policies explicit by ensuring a common understanding of critical concepts such as the DoD and DoR.
- collaboratively improving and evolving by adjusting WIP limits as needed.

A task board is a way to visualize the tasks to be completed in a sprint. It indicates the status and progress of tasks. A Kanban board is a variant of a task board that is used to visualize a workflow. The use of Kanban cards originated at a Japanese Toyota assembly plant; they were applied to exert control over the company's production processes and resulted in lead times being shortened by approximately one-third when compared to other similar plants (Poppendieck & Cusumano, 2012). The type and quantities of intermediate products to be produced, and those required for production processes, are recorded on signal cards and visibly displayed on boards; *Kanban* literally translates to the word "signboard" in Japanese (Tanner & Dauane, 2017, p. 181). Upstream stations then use the signal cards to determine what will be required in the near future. The wall or board where these cards are displayed is the Kanban board. This principle is applied to visualize and organize tasks for and within Agile Development Teams. All tasks of a cycle (or a sprint) are indicated on the Kanban Board, i.e., tasks still to be done in the first column, WIP in the second column, and completed tasks in the third column. It is also referred to as the pull system, as it "pulls" items from one column to the next as tasks are completed.

Tanner and Dauane (2017) explain that Kanban development applies the following concepts that should be defined as policies and outline the rules to be followed:

- the backlog, a prioritized list of work items still to complete.
- inclusion criteria, which defines the items that will be added to the backlog.
- done items, which are completed items.
- reverse items, which are items moving to a previous state.
- bottlenecks, which are items that limit progress and must be broken down into smaller items.
- performance measurement tools, e.g., burn down charts.
- validated learning, which involves measuring the value of a completed feature.
- waste, which are elements that do not add or produce value, defects, etc.
- stand-ups and meeting planning, which are used to discuss progress and next steps.
- feedback loops, which are used to obtain feedback in order to adapt when needed.
- an avatar, which visually represents a team member on the board.

Advantages of Kanban are similar to that of other Agile methodologies. It facilitates rapid development and deployment, focuses on the customer's requirements, and enhances communication and coordination. Insufficient experience and understanding of the process poses a challenge to successful implementation.

The lean development (LD) methodology was developed by Bob Charette (Anderson, 2012). LD is not exclusively used for software development; it is applied to achieve business value by means of business strategies and projects. It combines principles and concepts from risk management, as per Charette's experience, and lean manufacturing, as per the work of Womack, Jones, and Roos (Boehm & Turner, 2004). In an LD context, agility encompasses the ability to tolerate change. Accordingly, LD entails a three-tiered approach that is focused on change in order to achieve competitiveness. LD is a product-focused process, comprising of three phases, i.e., start-up, steady state, and transition renewal. Overall planning, including business cases and feasibility studies, are completed during start-up. The steady state phase involves iterations of designing and building. The transition renewal phase involves development and delivery of documentation, as well as the maintenance of the delivered product.

4.5 Scaling Agile Development

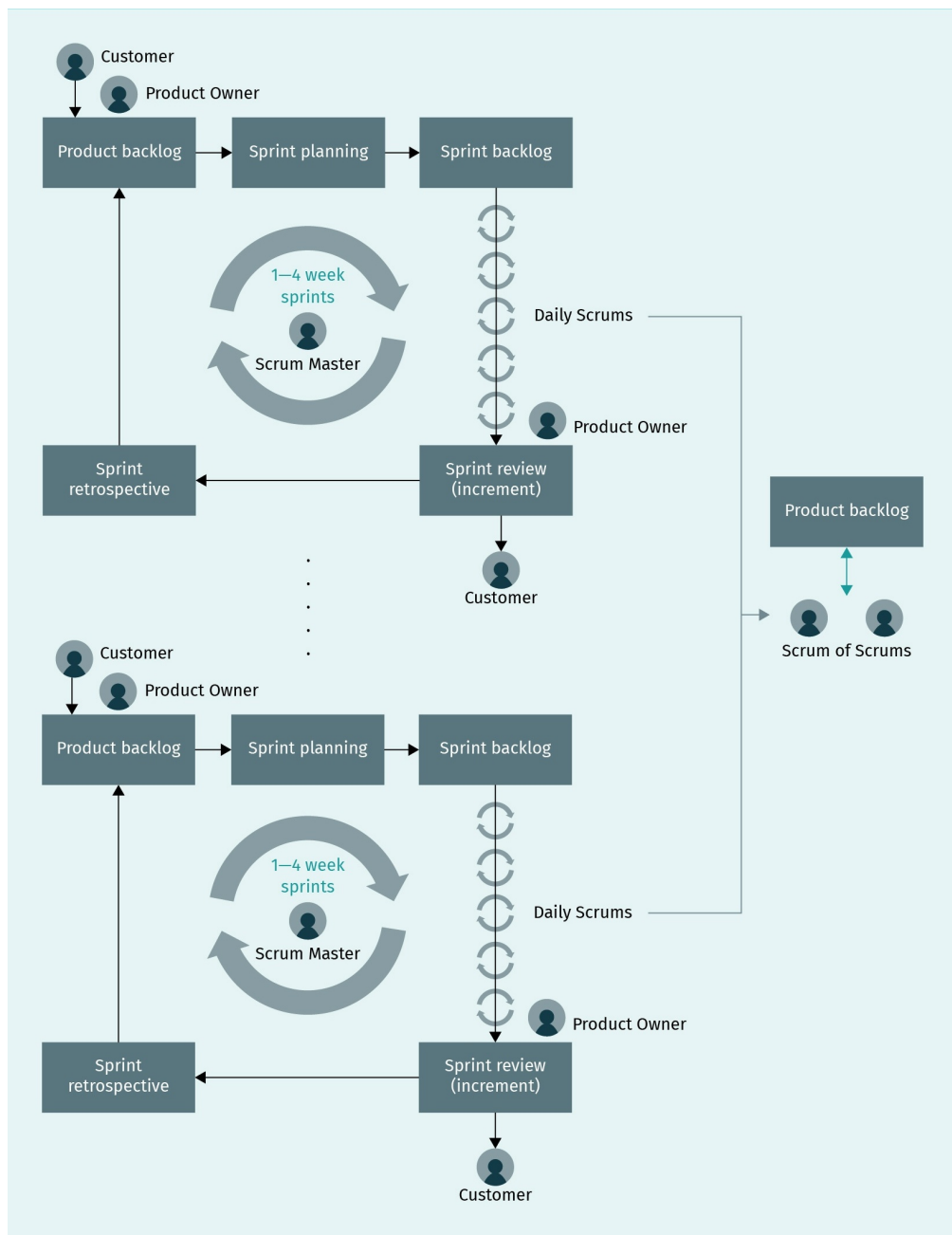
Dingsøy and Moe (2013) assert that although Agile methodologies were initially developed for small teams, they quickly became popular among larger teams. As the success rates of Agile development increased, project teams that work on larger or more complex projects (e.g., for projects that involve more than one team operating independently, teams distributed across geographical locations, or virtual teams) also started to consider tailored versions of them. This should theoretically result in increased customer satisfaction, rapid delivery, reduction in costs and overheads, and development of products and services that address rapidly changing needs. However, the application of Agile methodologies in large projects is significantly different and more complex than in small projects.

Dingsøy and Moe (2013) further explain that scaling Agile principles and practices poses challenges, e.g., longer planning horizons, levels of delegated authority, and synchronization of deliverables. Larger projects typically span over more than a year, requiring product-to-market roadmaps of 12–18 months, while the product backlogs for the teams are refined in shorter periods, typically between two and three weeks. This results in a disconnect regarding the level of refinement of the solution. The Development Team may work on details and finalize code, only to receive change requests from a higher level at a later stage, resulting in time-consuming and costly re-work or even the discarding of finalized code. The lack of management frameworks at higher levels, when compared to well understood and practiced Agile frameworks within the smaller teams, also poses a risk. Furthermore, levels of authority in larger and more complex projects interfere with practices, such as Scrum, where the Product Owner is responsible for the full life cycle of the product, including return on investment (ROI) and market performance, versus larger projects where responsibilities are segregated and project or portfolio managers may be jointly or individually responsible for different aspects. Disconnects such as these negatively affect the performance of small Agile teams, impose upon their autonomy, hinder the self-organizing dimension of the team, and are an obstacle to effective synchronization of delivery and integration of solutions. The context of application and decisions regarding modification of an Agile approach must be considered carefully when tailoring it to a large, complex project. This resulted in variants of Agile, e.g., the Scrum of Scrums (SoS), Large-Scale Scrum (LeSS), and Scaled Agile Framework (SAFe) emerging.

The Scrum of Scrums

Scrum of Scrums (SoS) involves dividing a project team into groups of Agile teams, where each team selects an “ambassador” to participate in frequent meetings with other selected ambassadors (Raps, 2017). The ambassador roles are typically undertaken by the Scrum Masters, but it can also be another team member. These meetings occur as frequently as necessary, and last approximately 15 minutes. The SoS discusses the status of each team, and also other issues or challenges that they may face. They typically discuss identified obstacles and ways to overcome them, issues regarding interfaces between separately developed solutions and integration of the complete solution, and responsibilities of and boundaries between individual teams. The SoS is presented below.

Figure 45: The Scrum of Scrums



Source: Jamsa & Harkiolakis, 2019.

Large-Scale Scrum

Large-Scale Scrum (LeSS) is the result of efforts to apply the purpose, elements, and elegance of Scrum to large projects. LeSS aims to enable organizational simplicity and purposefulness, but in a non-prescriptive manner (Larman & Vodde, 2017). Adapting to such a large scale, as is required by LeSS, encompasses profound organizational change. LeSS is

to be viewed as an organizational design framework, rather than a project management practice. LeSS includes many of the principles and ideas of Scrum. For example, it comprises a single product backlog, a DoD, a Product Owner, a sprint, an increment at the end of each Scrum, and cross-functional teams. Additionally, it entails two-part sprint planning, where part one involves a typical Scrum team, and part two is similar to SoS. LeSS differs from SoS in that it observes the large-scale view as a single Scrum, following the principles, rules, elements, and purpose of Scrum, rather than as another management level on top of individual Scrums. According to The Less Company B.V. (n.d.), LeSS has the following nine core principles:

1. “Transparency”, which is evident from tangible items that are completed (done), entailing “short cycles, working together cooperatively, common definitions, and driving out fear in the workplace” (para. 3)
2. “More with less”, which includes the concepts of “empirical process control: more leaning with less defined processes”; “lean thinking: more value with less waste and overhead”; and scaling through “more ownership, purpose, and joy with less roles, artifacts, and special groups” (para. 4)
3. “Whole-product focus”, which means having one product backlog, Product Owner, shippable product, and sprint, regardless of the number of teams involved. This is based on the fact that customers want valuable functionality in a single cohesive product, rather than technical components in separate parts.
4. “Customer-centric”, which involves learning about customers’ real problems and solving them by involving customers in meaningful feedback loops.
5. “Continuous improvement towards perfection”, which means striving to delight customers with perfect products, as well as improving the environment and lives, by continuously doing “humble and radical improvement experiments” (para. 7).
6. “Lean thinking”, which involves “respect for people and continuous improvement” (para. 9) by means of an organizational system aimed at eliminating waste, e.g., by reducing partially done work and delivering results as quickly as possible; simultaneously, decisions are made as late as possible to reduce uncertainty and while taking into account the continuously changing environment.
7. “Systems thinking” which entails exploring and optimizing the system as a whole (rather than only individual parts of it), and applying systems modeling techniques to explore system dynamics.
8. “Queueing theory”, which involves an understanding of how systems with queues will behave in the research and development (R&D) domain, and the application of the insights to manage aspects such as “queue sizes, work-in-progress limits, multitasking, work-packages, and variability” (para. 10).

The Scaled Agile Framework

The latest version of the Scaled Agile Framework (SAFe), i.e., SAFe® 5.0, entails seven core competencies that are applied in lean enterprises. The competencies relate to: lean-Agile leadership, a continuous learning culture, team and technical agility, Agile product delivery, enterprise solution delivery, lean portfolio management, and organizational agility (Scaled Agile, Inc., 2019). The competencies (with the exception of lean-Agile leadership) target different levels in the hierarchy, i.e., the Essential SAFe configuration, the Large Sol-

tion SAFe configuration, and the Portfolio SAFe configuration. Mastering all of this is critical to achieving and sustaining business agility and a competitive advantage in a modern marketplace. SAFe extends to guide both enterprises and governments.

Knaster and Leffingwell (2020) describe the competencies as follows: The lean-Agile leadership competency describes driving and sustaining organizational change. These leaders empower teams and lead by example to instill change. The continuous learning culture competency entails values and practices to encourage all to continue to increase knowledge, and also to constantly improve and innovate. Team and technical agility comprise the critical skills, principles, and practices applied to create solutions of high-quality for customers. Teams should remain productive and continue to deliver value. Agile product delivery refers to the approach that enables organizations to continuously define, build, and release products and services of value, in order to delight customers and remain competitive. The enterprise solution delivery competency describes the application of principles and practices to specify, develop, deploy, operate, and maintain large applications and systems. Lean portfolio management involves the alignment of strategy and execution through the application of lean and systems thinking approaches; it enables organizations to meet commitments while also continuing to innovate. Organizational agility refers to the ability to optimize business processes, properly evolve strategy, and rapidly adapt as and when required.

Knaster and Leffingwell (2020) state that all SAFe configurations are built upon the Essential SAFe configuration, which applies the principles and practices of the three core competencies: Agile-lean leadership, team and technical agility, and Agile product delivery. The Agile Release Train (ART) anchors SAFe. ART refers to an organizational structure that involves dedicating Agile teams, key stakeholders, and other resources to an ongoing mission. The Large Solution SAFe configuration is built upon the Essential SAFe configuration as it supports the development of large and complex solutions that require multiple ARTs and suppliers, and adds additional artifacts, events, roles, and coordination. It is implemented through a Solution Train, i.e., an organizational construct to facilitate development of large, multi-disciplinary, and complex software and systems. Large Solution SAFe adds the competency of enterprise solution delivery, over and above the core competencies of the Essential SAFe. The Portfolio SAFe can also be built on the Essential SAFe configuration; it signifies the minimum set of competencies and practices required for complete business agility. It provides three competencies in addition to the core competencies of the Essential SAFe: lean portfolio management, organizational agility, and continuous learning culture.

When all three configurations are applied to form the Full SAFe, it forms the most comprehensive configuration, including all seven competencies. Multiple instances of various SAFe configurations can also be used in an organization. The Spanning Palette is always indicated as part of the SAFe. It entails the specific elements, roles, and artifacts that an organization decides to include in a SAFe. The Spanning Palette is a selection of vision, roadmap, milestones, shared services, community of practice (CoP), system team, lean user experience (UX), and metrics. Each SAFe also includes a description of its foundation. It outlines the organization's selection, as applicable to them, of supporting elements that

they require to deliver value. These include the lean-Agile leaders, core values, lean-Agile mindset, SAFe principles, SAFe program consultants (SPCs), and an implementation roadmap (Knaster & Leffingwell, 2020).

4.6 Hybrid Processes

Agile and plan-driven approaches differ fundamentally in the way that decisions are made. Agile development is knowledge-driven, i.e., in Agile development, important decisions are based on knowledge gained as part of, and during, the project. On the other hand, traditional software development is assumption-driven; plans are based on assumptions and created in the run-up to the project or in very early phases. Decisions are also made based mostly on the current level of knowledge and on previous experience of those involved in the project. Both approaches have their advantages, disadvantages, strengths, and weaknesses. Both also have specific areas of application in which they are most suitable; however, it would be nearly impossible to align all software projects, and also the individual components of large and complex projects, to only one approach. Theocharis et al. (2015) summarize it by saying that the typical traditional processes “aim to address the whole software project lifecycle, e.g., by providing comprehensive guidelines, standardized procedures, project planning templates, and interfaces to further organization processes” (p. 150), whereas Agile methods aim to simplify software processes as much as possible in order to minimize rules, formalities, and administration.

Hybrid approaches combine different types of models or methodologies. These include, for example, approaches that combine plan-driven and Agile methodologies. Research has shown that companies apply content-specific hybrid approaches to develop software (Theocharis et al., 2015). Combinations of models that often emerge in practice are V-Modell XT and Scrum, where Scrum is then embodied using Kanban and Extreme Programming practices as well as the combination of the traditional waterfall, Scrum, and the Agile Unified Process (Theocharis et al., 2015).

The latter is also referred to as the Water-Scrum-Fall, which was introduced by Forrester Research, Inc. as a reality that organizations are faced with, rather than a methodology. However, it is now acknowledged as a useful hybrid approach, taking advantage of the best of both Agile and plan-driven development (Kneuper, 2018). It can include the application of the beginning and end phases of a waterfall model to analyze requirements, and do acceptance testing and deployment, but implements typical Scrum sprints in the middle phases, e.g., during the design, implementation, and testing phases.



SUMMARY

Process and tools are important, but cannot be valued over customer-centric principles, such as collaboration and communication. Well-defined process models and the use of tools can only be effective when applied collaboratively and when stakeholders cooperate to achieve a

mutual goal. Agile development approaches aim to overcome the challenges of traditional development approaches; however, in practice, both still have value. Since it is crucial to find the optimum blend, individual projects are often implemented using “hybrid” approaches.

Scrum is an evolutionary framework used to organize work in short cycles, after which it is carried out by teams. A team consists of a Scrum Master, a Product Owner, and developers. A product backlog is used to prioritize items for the sprint backlog. Items on the sprint backlog are completed within a single sprint, and progress is continuously discussed during daily Scrum sessions. Increments of solutions are inspected during the sprint review and adaptations are made, as needed, for the next sprint. Management tools, such as task boards and burn down charts, are used to visualize and manage the process.

Agile approaches achieve success and have therefore become popular. They are most suited for smaller scale projects, so attempts are made to scale them for larger and more complex projects. Examples of these include the SoS, LeSS, and SAFe. Patterns in the application of hybrid approaches emerged in the form of the Water-Scrum-Fall, combining the waterfall, Scrum, and Agile Unified Process, as well as V-Modell XT and Scrum, with Kanban and XP practices.

UNIT 5

THE SOFTWARE PRODUCT LIFE CYCLE

STUDY GOALS

On completion of this unit, you will have learned ...

- about customizable detailed-level processes.
- the value of information technology (IT) service management.
- to apply DevOps to streamline and integrate development and operations.
- about information, data security, safety, and privacy.

5. THE SOFTWARE PRODUCT LIFE CYCLE

Introduction

Software projects are implemented by means of software processes and life cycles. They facilitate systematic structuring and execution of applicable design, development, implementation, and maintenance activities, throughout the life cycle of software and software systems. The popularity of iterative and incremental development frameworks is increasing; they are believed to be more responsive to risks and ever-changing customer needs and market demands. However, many of these approaches are still relatively new and anecdotal, whereas large, integrated, and complex projects still require the meticulousness of traditional plan-driven and architecture-centric methods. Accordingly, hybrid approaches are often applied, and, in view of that, many organizations choose to adopt customizable iterative and incremental development frameworks, such as the Unified Process (UP) (Scott, 2002) and **German V-Modell XT** (Deutschland, 2004). They are modular and therefore adaptable; projects are tailored according to the characteristics and relative complexity of the envisaged project without compromising quality.

Organizations continue to depend on software and information systems to function effectively. As a result, information technology (IT) infrastructure, assets, and services are essential for modern-day businesses. They are also implemented and maintained for the benefit of customers and, accordingly, managed throughout their life cycles; they must be appropriately managed in order to ensure standardized, yet tailored, value for customers. The practice of **IT service management** (ITSM) ensures that IT services remain relevant and adhere to defined quality standards (Esposito & Rogers, 2013). ITSM is aided by frameworks, such as ITIL[®], formerly known as the Information Technology Infrastructure Library.

IT service management

These are management processes to ensure suitable and high-quality services, and IT assets that provide business value.

Silo mentality (a culture where different organizational departments choose to work in isolation, rather than collaboratively) costs organizations dearly. It reduces productivity, performance, and even staff morale. Similarly, different IT departments do not function optimally when they do not collaborate and communicate. In these cases, customer requirements are not sufficiently met. So, a new and innovative way of working was devised within the software engineering realm, i.e., **DevOps** (the unification of development and operations). It aims to effectively automate collaborative processes between teams. The outcome is improved communication and higher quality solutions and enhanced deployment speed, frequency, and reliability (Mishra & Otaiwi, 2020; Perera et al., 2017). DevOps aims to resolve human challenges by effectively utilizing technology and enabling adoption of Agile and lean practices.

DevOps

This is a set of practices that streamlines and automates processes between software development and other IT teams.

Software, regardless of its area of application, must be developed in a manner such that it ensures the safety, security, and privacy of customers and users. In addition, data and information are valuable organizational assets that must be properly protected in order to remain secure. Hence, all applicable standards and regulations must be adhered to. Extensive instructions regarding information security have been documented and published by

the German Federal Office for Information Security (BSI). The *IT-Grundschutz* catalog provides a framework for the management of IT security; it describes aspects that can threaten security and response measures (ENISA, n.d.).

5.1 Detailed-Level Process Models: Unified Process and V-Modell XT

Unclear and changing requirements in the software domain resulted in the adoption of iterative and incremental development frameworks that are customizable, as per the needs of the project. The Unified Process (UP) and the German V-Modell XT are examples of modular and customizable frameworks.

The Unified Process

When applying UP, a project is planned and executed according to four core phases: inception, elaboration, construction, and transition. Phases are typically divided into multiple iterations (Scott, 2002). Additional phases, such as production and disposal, can be added if required. Most iterations will include general activities, such as requirements gathering, design, implementation, and testing; the relative effort and emphasis of these will, however, differ as the project progresses.

During the inception phase, a business idea is considered and elaborated upon to produce a project. This phase is comparable to a feasibility study—goals are outlined, the scope is defined, and an initial schedule and cost estimate are prepared. Furthermore, a business case is formulated based on core requirements, key features, and constraints (Scott, 2002). This phase ends with the life cycle objective milestone.

Throughout the elaboration phase, project requirements are elicited and potential risks are identified so that the most critical risks are addressed as early as possible in the process. Risks are identified for mitigation in order to prevent project failure. The core elements of the system to be developed are also identified; the level of detail must be sufficient so that developers can conceptualize ideal solutions. In this phase, the system architecture is established and validated. Tools, such as the Unified Modelling Language (UML), are used to explore and model the architecture views of solution components. Additionally, they implement package diagrams, object-oriented class hierarchies (which are only implemented during development), and use cases (which identify the actors and ways that they will be interacting with the system) (Arlow & Neustadt, 2002). The aim of this phase is to demonstrate a stable architecture that supports key functionalities and behaves suitably in terms of, e.g., performance, scalability, and cost. It is validated by implementing an executable architecture baseline, i.e., a partial implementation of a system that includes its most significant components (Scott, 2002). This phase provides a detailed plan for the next phase.

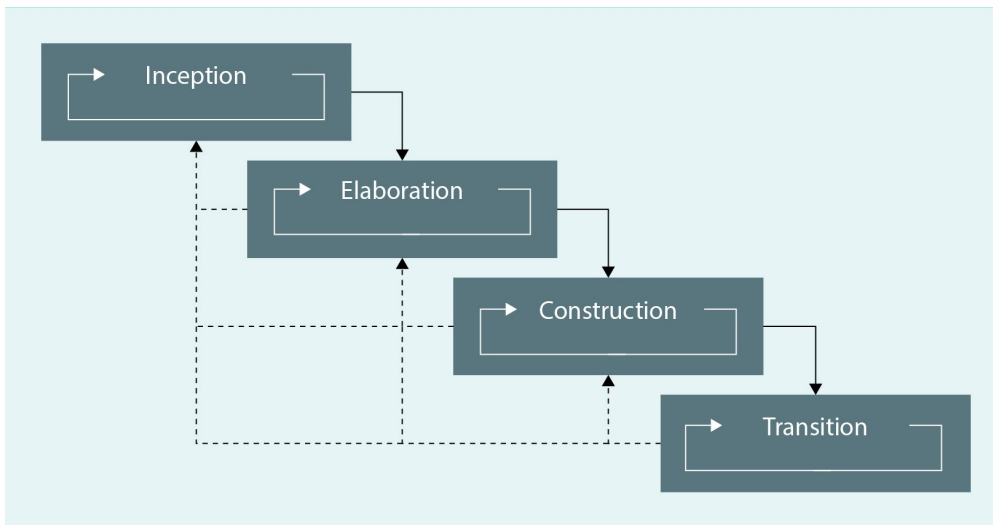
In the construction phase, developers aim to establish a high-quality solution based on the foundation established during the elaboration phase. UML activity diagrams, sequence diagrams, collaboration diagrams, etc., are used during this phase. Developers write, test, and debug code, with a focus on creating high-quality deliverables that are thoroughly tested and ready for user acceptance upon release. This phase ends when the software system is ready for deployment (Scott, 2002).

The solution is deployed in the transition phase (Scott, 2002). In this phase, requirements for infrastructure (e.g., computers, servers, and networks) are fulfilled, documentation is completed, and users are trained. The solution will be in production, and users can begin to use it. The solution is handed over to a maintenance team for the management of fixes, upgrades, and customer requests. Based on initial feedback from the customer, the project may temporarily revert (i.e., iterate) back to previous phases to address serious issues.

When required, additional phases are considered in order to address other project aspects and the remainder of the full software life cycle, e.g., a production phase may be implemented after the transition phase to determine how users respond to the solution. Feedback from this phase may be useful for upgrades or for future projects. A disposal phase may also be considered, where users transition to a system replacing other systems to ensure minimal disruption to the users and their organizations.

Unified process (UP) phases typically iterate several times to address phase-specific issues, challenges, and obstacles. The exception here may be the inception phase, as it will typically not iterate for smaller projects (it is relatively abstract by nature); however, in large complex projects, the inception phase can also be divided into iterations. Risks are often identified during elaboration and then refined for mitigation during the subsequent iterations of the elaboration phase. Similarly, during the first construction phase, the focus will be on developing critical and important features, and the remaining features will be added during subsequent iterations. UP iterations are shown in the figure below.

Figure 46: Unified Process Iterations

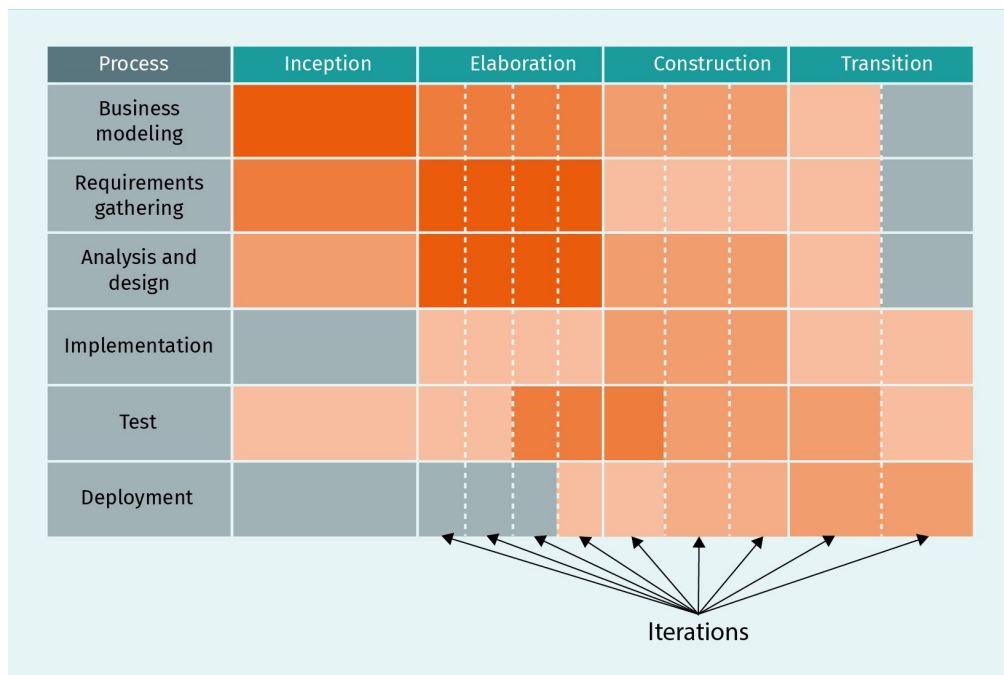


Source: Jamsa & Harkiolakis, 2019.

Even though UP is highly adaptable, it can be relatively complex to customize it for a specific organizational context and project, as the defined UP process does not have a guide for tailoring. Customization can also be difficult to repeat for a similar project (Ruiz et al., 2018). There are a number of variants of UP: OpenUP, Enterprise Unified Process (EUP), Agile Unified Process (AUP), and Rational Unified Process (RUP). The most popular variant of UP is the Rational Unified Process (RUP), which applies the core life cycle phases of inception, elaboration, construction, and transition. It also groups tasks according to six core disciplines (also referred to as workflows), i.e., business modeling, requirements gathering, analysis and design, implementation, testing, and deployment (Kruchten, 2004). RUP has advantages over other unified processes; namely, it is owned by and included in an IBM product (the IBM Rational Method Composer (RMC)), and IBM provides useful tools, such as artifact templates, that can be used for any intermediate or final result produced by the project. The artifacts are useful for visualization, modeling, document management, tracking of change requests, and performance profiling. The popularity of RUP has increased for large projects as a result of these development tools. However, it also resulted in a significant disadvantage, since it is now difficult to apply RUP when one does not have access to these tools.

A hypothetical case that incorporates RUP is given in the figure below, which shows how it supports resource planning and workload balancing. It illustrates that business modeling will require more effort (as indicated by the bright orange color) at the inception phase, but this effort will gradually reduce in the remaining phases. The deployment phase requires little effort in the beginning, and more towards the end of the project. This figure also indicates the multiple iterations within each phase, with the exact number being defined as part of the project planning.

Figure 47: Workload Distribution in UP Phases



Source: Jamsa & Harkiolkakis, 2019.

Furthermore, an RUP approach incorporates best practices from software engineering in order to eliminate errors and increase productivity. These include the following:

- using iterative development to enable and accommodate changing user requirements;
- decomposing large and complex projects into smaller components to be independently tested before assembly into a larger system (and also for re-use);
- using tools, such as UML diagrams, that enable visualization of components and their interactions to clarify work to be done and make the roles of various actors visible; and
- continuously integrating various components developed by different teams (that are either collated or distributed) in order to eliminate possible side effects resulting from the different development tools and configurations used.

V-Modell XT

V-Modell XT
A tailorable model (divided into process modules), the V-Modell XT describes roles and activities to be performed in order to create work products.

The **V-Modell XT** enables flexibility and tailoring (the XT stands for extreme tailoring). It is designed to guide tailored planning and execution of system development projects by defining project results, i.e., products. The model is divided into a number of process modules describing the roles, responsibilities, procedures, and activities that must be performed to create these results (Microtool, n.d.). The user selects only the applicable modules, according to the characteristics of the project, and its tailoring guide facilitates the creation of a set of adapted procedures whereby to execute the project based on the applicable project characteristics (Kneuper, 2018). It is a state-of-the-art software process and life cycle model initially developed for the German government for use with large

projects, but it is adaptable for any size project. It includes a meta-model, a model, and tool and documentation (template) support. The V-Modell XT supersedes and replaces its previous version, V-Modell 97.

On the meta level, the model is organized into different packages. The different packages and their purposes are as follows (Kneuper, 2018):

- The base package describes the structure and manner in which the model is documented, e.g., its chapters and tool references, as well as its method.
- The statics package explains all of the process modules, including the basic components of each module and the relationships and dependencies of the components.
- The dynamics package states the ordered sequence of all the activities, as defined in the statics package.
- The adoption package provides standardized criteria that define the tailoring of the model.
- The mapping to standards package enables references to compatible current standards and regulations (e.g., ISO9001:2000, ISO/IEC 15288, and the Capability Maturity Model Integration (CMMI®)).

The V-Modell XT is based on the V-Model for effectively guiding the planning and execution of a project throughout its entire life cycle and aims to describe “in detail ‘who’ has to do ‘what’ and ‘when’ within a project” (Deutschland, 2004, p. 5). In addition, it offers traceability and delivers high-quality and reliable results. The model aims to minimize project risks, improve and guarantee the quality of solutions, reduce the cost of a project, and improve communication between stakeholders, customers, and contractors. It implicitly demands and mandates structured management and control over requirements and changes, but does not explicitly prescribe how to do so (Microtool, n.d.).

According to the V-Modell XT, projects are classified according to two characteristics: the subject of the project and the project role. The subject is based on the purpose of the project, namely, whether the project will be to develop a system or an organization-specific process model. The project role differentiates between the role of a customer and a supplier. Furthermore, it also distinguishes between different types of projects, and provides suitable project execution strategies for the different types. The types of projects include the following (Deutschland, 2004):

- a system development project from the perspective of a customer,
- a system development project from the perspective of an external supplier,
- a typical internal system development project where both the customer and supplier belong to the same organization, and
- the introduction and maintenance of an organization-specific process model.

Project execution strategies include sequences of activities and tasks, work products to be developed, and the relevant roles required to create the products—these are defined in process modules. A project execution strategy also comprises a series of decision gates, indicating the stages where project progress should be evaluated. At the evaluation points, a project is given a go-ahead to progress further, corrective actions are initiated, or it is canceled. Project execution strategies involve compulsory process modules and deci-

sion gates, as specified for each type of project, as well as optional process modules and decision gates. The execution of a project can be tailored based on the actual conditions of the project; a user is able to select one strategy, or a combination of project execution strategies, applicable to and associated with the project type (Deutschland, 2004). The V-Modell XT does not provide guidance for operational use, maintenance, or decommissioning of systems.

In addition to providing tailoring options, this model also supports both V-shaped and iterative and incremental development. It is an Agile, yet plan-driven, approach that provides visibility with regard to project progress and status, and results in high-quality solutions. It enables explicit reflection on and incorporation of all applicable processes of both the customer and the supplier.

5.2 IT Service Management and Operations

Gartner, Inc. (n.d.) defines information technology (IT) services as “the application of business and technical expertise to enable organizations in creation, management and optimization of or access to information and business processes” (para. 1). Gartner, Inc. (n.d.) distinguishes between categories of IT services, i.e., “business process services, application services and infrastructure services” (para. 1), and argues that different skills are required at different stages of the service’s life cycle, i.e., dependent on whether it is still in “design” mode; being developed (“build” mode); or operational (“run” mode). Therefore, suitable strategies are required to manage different IT services throughout their life cycles.

Nowadays, IT services are an intrinsic and key component embedded in modern business; they are no longer merely supplemental services to enable and support business processes (Esposito & Rogers, 2013). Therefore, IT services must be implemented and managed aptly. ITSM ensures that applicable and high-quality IT services are provided and managed appropriately and IT assets continue to provide business value. ITSM aims to provide customers with a range of suitable and valuable IT services that are standardized, yet scaled and tailored to their requirements (Thejendra, 2014). It involves understanding the customer’s needs and managing expectations; standardization of IT services as and where applicable; the establishment and regulation of IT processes, tools, and roles; the measurement and evaluation of services; and the optimization of services. Traditionally, ITSM efforts seem to call for structure and long-term planning; they involve continuous and longstanding change management and improvement projects. However, ITSM can also be effectively implemented in an Agile manner when improvements are prioritized in the short-term and made iteratively by responsive and collaborative business and IT teams (Ferris, 2017).

Understanding Customer Needs and Managing Expectations

Exploring customers' needs can be compared to requirements gathering, analysis, and engineering. IT service providers aim to understand customers' actual requirements in terms of the IT services to be provided (Esposito & Rogers, 2013). ITSM differentiates between functional and quality requirements. Functional requirements describe required technical functions and services. The quantifiable quality requirements that these functions and services must adhere to, as well as the actions that must be performed to reach required levels of quality, are described and defined in service level agreements (SLAs) and operational level agreements (OLAs). SLAs and OLAs typically describe and underpin contractual obligations of external service providers, or, when IT service providers are internal to the company, the acceptable (minimum) levels of services that the business can tolerate (Thejendra, 2014).

SLAs define the required quality criteria for functions and services (Esposito & Rogers, 2013). They are written in non-technical (business) terms and define, e.g., when IT services must be available; the number of supported users; guaranteed simultaneous access to a service; guaranteed reaction, response, and resolution times in case of minor and major issues or failures; and backup strategies. The significance and critical nature of the IT services, typically measured in terms of potential harm (economic or otherwise) caused by the non-availability thereof, determines the levels of quality that are agreed upon in SLAs. SLAs must be outcome-based and facilitate collaboration between the teams working together to achieve collective goals (Macdermid, 2019). The criteria must be measurable, and they can be differentiated. For example, an email service can be made available to staff during specified working hours only, but it must also be available after hours and over weekends for executive management. Higher service levels typically result in higher overheads and costs, meaning that optimum service levels must be determined for the business. Free-market IT services (e.g., online and cloud storage or free email) may be sufficient for private users, but the lower and unguaranteed levels of these services make them inadequate for corporate users.

OLAs are internal and technical service provider agreements that support SLAs and underpin contracts (Thejendra, 2014). OLAs describe the technical functionality and actions that will be required and employed to deliver services at required levels of quality, and are accordingly written in technical terms. They outline roles and responsibilities of stakeholders and IT teams, as well as resources (tools and infrastructure) required to provide and support the services. OLAs define, e.g., what a server team will do to patch servers, required server memory, number of servers, bandwidth, CPU speed, and reliability and redundancy of hardware and resources.

Standardization of IT Services

Technical and organizational complexity of providing IT services can be significantly simplified by standardizing the provision of IT services and assets; "standardization is essential to achieve both efficiency and effectiveness" (Esposito & Rogers, 2013, p. 144). Standardization of IT services involves a standardized service catalog detailing the available categories of services and corresponding levels of services to customers. Customers

choose from the service catalog according to their needs; if the services provided are insufficient, suitable procedures should be in place to motivate expansion or modification of the catalog services.

Standardizing specific assets and products (e.g., limiting customers by providing them with a limited selection and range of printers, laptops, and desktop computers) makes it possible to leverage economies of scale when purchasing assets. In addition, it becomes possible to establish and foster favorable long-term relationships with suppliers when purchasing larger quantities of certain products, or purchasing more frequently from them. It may also be possible to negotiate better deals and favorable support agreements with preferred suppliers. In addition, having a limited range of standard products to maintain will reduce costs.

Establishment of IT Processes, Tools, and Roles

Process model frameworks for ITSM include globally-established best practices to manage delivery and support of IT services and assets. The ITIL framework is an example of such a practice that is widely accepted and applied. It guides the management of IT services and is used by organizations “to run their business from strategy to daily reality” (Joret, 2019, p. 2).

The IT Infrastructure Library

ITIL was developed in the 1980s by the British Government’s Central Computer and Telecommunications Agency (CCTA) to manage their services (ITIL Central, n.d.). The initial version consisted of a library of books, each describing a process. Later on, ITIL V3 described 26 processes in support of four organizational functions and aligns with ISO/IEC 20000, which is the internationally recognized standard for service management. ITIL is the intellectual property of AXELOS, which is a joint venture between an administrative body of the UK government, i.e., the Cabinet Office, and Capita Plc. ITIL V3 is organized around the service life cycle, i.e., service strategy, service design, service transition, service operation, and continual service improvement (Joret, 2019). These are defined as follows:

- Service strategy includes policies and objectives to ensure business value; it is realized through practical decisions and proper planning.
- Service design ensures that services are built, evolved, operated, or withdrawn, as per the strategy, and taking all stakeholders into consideration.
- Service transition entails planning and management to implement services, ensuring quality and the satisfaction of stakeholders.
- Service operation involves repeating activities for ongoing support.
- Continual service improvement applies methods from the practice of quality management to continually improve.

The latest version, ITIL 4, is a holistic approach that is still evolving. It continues to consider established practices in a much wider context and embraces new ways of working (Joret, 2019). It now uses modern practices, such as Agile, lean, and DevOps. ITIL 4 supplements and complements ITIL V3, which provided detailed process descriptions that are still useful. However, ITIL 4 can also be used without applying the somewhat rigid and

complex process descriptions of ITIL V3. In line with agility, organizations can define their processes in a manner as simplified (or complex) as they required. Hence, ITIL 4 describes principles, concepts, and practices, rather than comprehensive and prescriptive process specifications. Furthermore, ITIL 4 refers to key activities, as well as essential inputs and outputs.

Joret (2019) explains that ITIL 4 integrates four key dimensions to effectively manage services: organizations and people, information and technology, partners and suppliers, and value streams and processes with components from the service value system (SVS). The SVS consists of generally applicable components that aim to facilitate creation of value using services that are enabled by IT. This includes guiding principles, continual improvement, and governance and practices guiding the central service value chain that is also linked to both value and opportunity or demand. ITIL 4 comprises a total of 34 practices that encompass three categories: general management, service management, and technical management (Joret, 2019). ITIL has been adopted by various and diverse small companies, as well as large industries, such as IBM, Microsoft, Sony, Toyota, HP, and various financial and banking institutions; it is adaptable and can be used on its own, or in conjunction with other practices for governance, quality, and architecture management, e.g., Agile, lean, COBIT® (formerly known as the Control Objectives for Information and Related Technologies), Six Sigma, and The Open Group Architecture Framework (TOGAF) (Arraj, 2013).

Measurement and Evaluation of Services

IT services must be measured and evaluated to ensure that the business knows how well its services are functioning, and where problems may arise (Esposito & Rogers, 2013). Service providers should be able to constantly assure their customers that they provide services at agreed upon levels, maintain services and assets, and resolve incidents and problems within agreed upon resolution times. Failure to keep to SLAs and OLAs will typically result in penalties for non-compliance. In addition to actual non-compliance, perceived quality of service, according to business users, should also be measured. If the measured service quality differs significantly from perceived service quality, it indicates a perception gap that must be discussed and bridged.

Optimization of Services

IT service operations should be optimized and managed to ensure high-quality services, management, and mitigation of contractual and SLA or OLA violations, and an optimal balance of costs versus benefits (Esposito & Rogers, 2013). Continuous measurement, evaluation, and reporting of findings aim to ensure continuous delivery of high-quality services. In addition, non-conformance and service level violations should be monitored and appropriately managed by determining underlying causes of failures, incidents, problems, and malfunctions, and implementing suitable corrective actions. It is also important to regularly reflect on the efficiency of costs versus benefits of IT services. If practical, standardized service provision can be streamlined and simplified, and it will enable suitable adaptation to new requirements, market conditions, and beneficial technological advancements.

5.3 DevOps

Various organizational IT operations and departments often prefer to work in silos, rather than collaboratively; this results in diverse and even conflicting leadership approaches, performance metrics, and, unfortunately, lack of awareness regarding important changes that impact downstream and across various teams. Patrick Debois mentioned at the Agile 2008 Conference that development and operations teams should improve interactions, and also referred to the need for Agile infrastructure (Debois, 2008). Following that, the DevOps (combining the terms development and operations) model aims to improve quality while simultaneously increasing deployment speed, frequency, and reliability (Mishra & Otaiwi, 2020; Perera et al., 2017). DevOps relies on strong collaboration between teams; it aims to resolve human challenges, rather than technical challenges, and focuses on rapid delivery by adopting Agile and lean practices and utilizing appropriate technology. DevOps includes the entire continuous integration, delivery, and deployment pipeline, i.e., from committing to a change that will be made to the code to post-deployment testing.

The benefits of DevOps are

- improved responsiveness to the demands of markets and customers;
- improved collaboration and increased trust between IT functions, e.g., IT operations and IT development teams;
- an environment that enables more frequent software releases, hence faster code delivery and time-to-market of solutions;
- faster software releases to production, with the software remaining of high quality and containing fewer errors and bugs; and
- decreased time to resolve vulnerabilities, bugs, and errors.

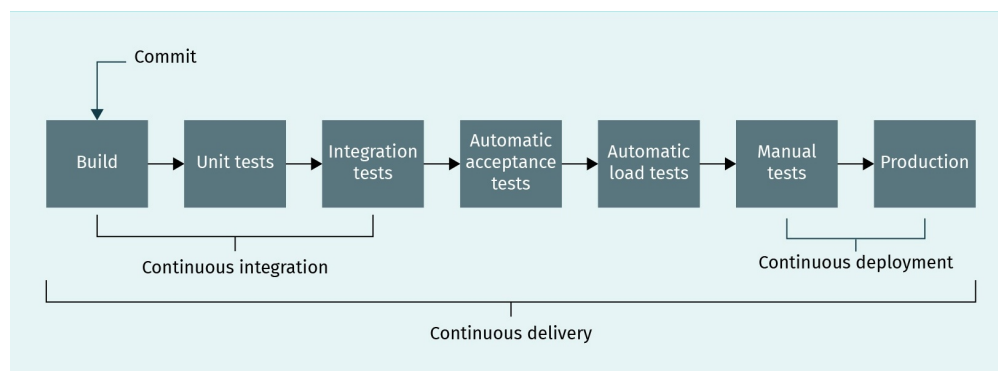
DevOps focuses on the human challenges, i.e., what should be accomplished, while continuous integration, continuous delivery, and continuous deployment are used to implement automation, i.e., the technical “how” of DevOps. Automation solutions are applied to technical challenges. The goal is to make deployment predictable, i.e., a routine process that can be performed on demand. It aims to produce higher quality solutions and, since errors can be identified and resolved early in the process, also reduce risks when releasing software.

Continues Integration, Deployment, and Delivery

The continuous delivery concept originated as part of Agile software development and commits to the first priority referenced in the Agile Manifesto, namely, to satisfy customers with valuable software that is delivered early and continuously (Beck et al., 2001b). Continuous integration is also not a new concept; it was applied in the 1990s in Extreme Programming. It refers to integration of development work continuously and frequently (at least daily, but preferably more often) (Red Hat, Inc., n.d.). Each small change that is completed is integrated into a new build and tested. Developers must check in their changes at least daily. Continuous delivery and continuous deployment include building, configuring, packaging, and uploading completed software to a repository in a way that supports frequent releases, as well as the automatic release of changes from the repository to produc-

tion (Red Hat, Inc., n.d.). Ultimately, all concepts, methods, and tools are needed to facilitate the continuous integration and the continuous delivery and deployment of software to a customer. A continuous delivery pipeline (sequenced, and mostly automated, activities that commence with “commits”, i.e., changes made to the source code) ends with the installation and configuration of the solution, and follows through with a series of largely automated integration and test phases (illustrated below).

Figure 48: The Continuous Delivery Pipeline



Source: Brückmann, 2015.

A continuous delivery pipeline differs from a traditional software process because the batch sizes of Agile projects are considerably smaller; developers only bring in a new function realized within a sprint, rather than a whole component or application (Mukherjee, n.d.). Also, the order in which testing takes place may be confusing for a traditional developer or a developer without Agile experience. In a continuous delivery pipeline, automated acceptance does not occur, as per the International Software Testing Qualification Board (ISTQB) standard, prior to the solution going live, as automated acceptance tests are done immediately after integration.

DevOps Principles

DevOps is based on three principles: flow, feedback, and continual learning and experimentation (Kim et al., 2016). Flow addresses fast delivery of work. It is achieved, for example, by making current and future work visible on task or Kanban boards, limiting the amount of work that is in progress at one time and reducing batch sizes. Feedback is implemented by continuously informing Development Teams about problems experienced by operations. Continual learning and experimentation can be enabled by fostering an organization-wide learning culture.

Challenges of DevOps

DevOps provides a multitude of benefits, but also poses some challenges. For example, it is not yet sufficiently standardized; the meaning and fundamentals of DevOps are not yet widely understood and accepted. Lack of knowledge about required supporting processes, infrastructure, methods, and tools can result in choices that do not support the organizational vision. It is also very challenging and expensive to set up, integrate, and

maintain tools. It remains difficult to automate testing and analysis sufficiently, so that no manual intervention is required. Teams may remain isolated, regardless of efforts to unite them, due to the very different nature of their work and goals in the organization. Deep-rooted cultural changes, such as what is required to effectively implement DevOps, are difficult to implement and maintain in large and established organizations. These challenges should not deter organizations from implementing DevOps; these are merely aspects that should be considered in depth during the planning and implementation processes. Integration of a DevOps culture with governance and enterprise architecture frameworks should be considered carefully.

5.4 Safety, Security, and Privacy

Software processes should address the safety, security, and privacy of software solutions in order to protect the customers and users thereof. Safety refers to the fact that a software system may not harm humans, material assets, or the environment. Security refers to confidentiality, availability, and integrity of data and information; it includes secure authentication and authorization, secure configuration and data transmission, secure storage, and secure access. Privacy refers to the appropriate use of data and information, and defines and governs the use of personal data. Safety, security, and privacy are ensured by appropriate governance and regulatory processes. Software processes must adhere to applicable ISO and IEC standards, and country-specific standards and regulations that relate to safety, security, and privacy.

Organizations' data, information, and systems must be appropriately managed and monitored to ensure that all of their data and information assets remain secure. The German Federal Office for Information Security (BSI) documented and published extensive instructions for information security—the *IT-Grundschutz* catalog, formerly known as the IT Baseline Protection Manual, provides instruction on information security and is compatible with the international standard ISO/IEC 27001 (German Federal Office for Information Security, n.d.). *IT-Grundschutz* was first released in 1994 and last updated in 2005. It provides a framework for the management of IT security and details aspects that can threaten security; it includes information about commonly used IT components and describes relevant threats and measures to counteract them. It enables the establishment of an Information Security Management System (ISMS) and is categorized by the European Union Agency for Cybersecurity as a method that supports risk assessment (RA) and risk management (RM) (ENISA, n.d.). In terms of RA, it includes risk identification, risk analysis, and risk evaluation. In terms of RM, it comprises risk assessment, risk treatment, risk acceptance, and risk communication.

ENISA (2020) explains that RA and RM are supported in *IT-Grundschutz* as follows:

- Risk identification supports RA as per the list and classifications of typical threats.
- Risk analysis supports RA as per the detailed description of each threat.
- Risk evaluation uses damage scenarios to assess exposure within an assessment of protection requirements.

- Risk treatment supports RM as per recommended safeguards that are cataloged and detailed.
- Risk acceptance is based on the description of how to handle threats in *IT-Grundschutz*.
- “Risk communication is part of the module ‘IT security management’ and especially handled within the safeguards S 2.191 ‘Drawing up of an Information Security Policy’ and S 2.200 ‘Preparation of management reports on IT security’” (ENISA, n.d., Identification section).

According to ENISA (2020), the process of establishing an Information Security Management System (ISMS), using *IT-Grundschutz*, includes two primary steps:

- Initialization of the process is the first step and involves
 - definition of IT security goals and business environment,
 - establishment of an organizational structure for IT security, and
 - provision of necessary resources (ENISA, n.d., Brief description section).
- Creation of the IT security concept involves
 - IT-structure analysis,
 - assessment of protection requirements,
 - modeling,
 - IT security check,
 - supplementary security analysis,
 - implementation planning and fulfillment,
 - maintenance, monitoring, and improvement of the process, and
 - *IT-Grundschutz* certification (optional) (ENISA, n.d., Brief description section).

Software processes should be enriched to ensure safety, security, and privacy, and certain steps can be taken in this regard (Kneuper, 2018). These include performing a risk analysis to identify, classify, and mitigate potential risks; determining relevant requisites, based on identified risks and any additional related requirements that the customer may have; defining processes to implement the requirements; and verifying of the implementation thereof. These should be included in the planning phases and progressively expanded upon so they become an inherent part of the configuration of the final version of the software—the properties related to safety, security, and privacy cannot be added later.



SUMMARY

Iterative and incremental software processes are increasingly used. Organizations want to be more responsive to changing demands and reduce risks while, at the same time, improving the reliability and quality of software solutions. Lighter and more Agile methods are often applied in conjunction with plan-driven approaches in order to reap the benefits of both. Accordingly, customizable iterative and incremental approaches are applied to large and complex projects that require both agility and high levels of quality. Examples of these are the Unified Process and the V-Modell XT.

IT infrastructure, assets, and services are an intrinsic part of a modern-day business. When these are not available, or do not meet the requirements of the business, the resultant effect can be devastating for the business. Therefore, these services must be appropriately tailored and managed to ensure value, and ITSM practices are an integral part of organizational management.

Contemporary practices, such as DevOps, are widely adopted and implemented to streamline ways of work and improve collaboration among different teams. DevOps relates to a culture that advocates and fosters collaboration, cooperation, communication, and reduction of risks. It aims to increase frequency of releases and enhance quality and reliability. It enables frequent, continuous, and automated integration, deployment, and delivery of small releases.

Software solutions must ensure the safety, security, and privacy of customers. Software processes must therefore ensure that software adheres to applicable standards and regulations. Also, IT assets and information services must be protected and remain secure. The *IT-Grundschatz* catalog provides a detailed framework for the management of IT security, describing aspects that can be a threat, as well as measures to counteract such threats.

UNIT 6

GOVERNANCE AND MANAGEMENT OF SOFTWARE PROCESSES

STUDY GOALS

On completion of this unit, you will have learned ...

- to apply governance principles within an enterprise architecture framework.
- how to design and deploy processes.
- which aspects to consider to effectively tailor processes.
- about applicable measures and tools used to assess and improve processes.

6. GOVERNANCE AND MANAGEMENT OF SOFTWARE PROCESSES

Introduction

Data and information are the currency of the twenty-first century. Accurate and appropriate information, when reliably provided, is indispensable in the new economy and for modern businesses (IT Governance Institute, 2005). Integrated and interrelated information technology (IT) and software systems enable this. Technological platforms drive businesses, and vice versa; they must therefore be managed and governed appropriately, collectively, and cooperatively through both business and IT leadership.

The management and governance of processes extends beyond software processes and models. It involves ensuring that IT assets and services add value and that IT processes are embedded and used as intended. It also entails highly interdependent IT and business systems and processes. Therefore, suitable and unified corporate, enterprise, and IT frameworks are applied. Frameworks, such as The Open Group Architecture Framework (TOGAF), guide the development of suitably tailored enterprise frameworks (Josey, 2018). IT processes, services, and assets must also be managed and measured in the context of organizational strategic goals. For this, COBIT® (previously the Control Objectives for Information and Related Technology) is widely used to facilitate IT governance and management (Dziak, 2019). Agile governance is still a new concept, but it is on the rise; it relates to the way management and governance frameworks are applied. In practice, business agility requires the effective employment of both Agile and governance capabilities (Luna et al., 2014).

Organizational and IT processes should be aligned, designed according to the needs and objectives of the organization, and properly deployed to ensure that they provide their intended value. Aspects such as continuously growing market demands, evolving customer requirements, and technological developments often necessitate process tailoring. However, this can be challenging and risky, so specific factors must be considered to ensure effective process tailoring (Xu & Ramesh, 2008).

Processes are designed and deployed according to process models. They should be continuously assessed in terms of quality, performance, and effectiveness. Improvement actions must also be identified, implemented, and monitored. To this end, Software Process Improvement (SPI) is applied; the SPI Manifesto defines values and principles to achieve SPI (EuroAsiaSPI², n.d.). The Capability Maturity Model Integration (CMMI) Institute provides guidance in terms of maturity models to evaluate the quality of processes and suggest areas to improve. CMMI models provide a roadmap to achieve process capability and maturity (ISACA, n.d.-c). Support tools are applied to simplify the complex tasks of process modeling, management, and enactment.

6.1 Process Governance

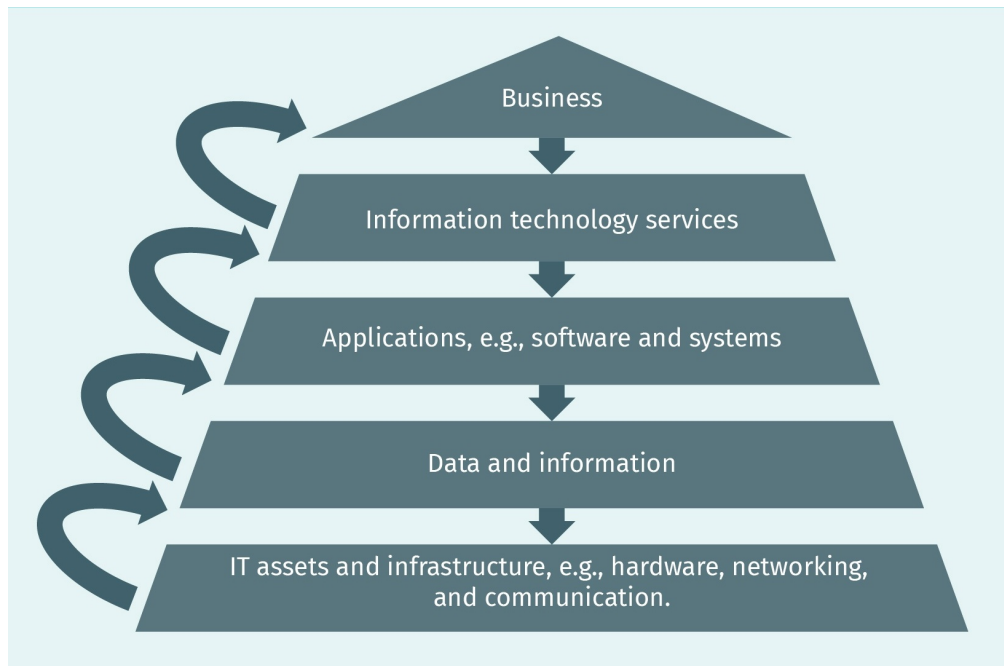
Organizational processes that are practical, fit-for-purpose, and manageable deliver business value. Accordingly, business processes must be managed and governed within suitable frameworks to ensure that they add value. IT architectures are an essential part of business strategy and an aspect of modern businesses that impact competitive edge (Gellweiler, 2020). Hence, organizations spend substantial resources to effectively manage and govern IT processes. IT management and governance are proven practices, and there are established frameworks in this regard. In addition, software development governance is a relatively new concept, which also aims to facilitate alignment between IT and business (Dubinsky & Kruchten, 2009; Juiz & Colomo-Palacios, 2020). For this to work, businesses and IT leadership must work together to create one overarching corporate and business architecture. IT leadership must understand business goals and objectives so that they can provide tailored processes, resulting IT assets, infrastructure, and artifacts to the business. Appropriate tailored software and IT processes must therefore be managed within a suitable structure. The **enterprise architecture** describes principles, practices, and parameters that can be used to design and construct business applications and systems. This is driven by the business architecture, which functions as “a single unifying concept” that “dictates the shape of the IT environment and supports effective IT governance” (IT Governance Institute, 2005, p. 7). It defines and explains all relevant governance structures, required business information, and associated and interrelated business and IT processes (The Object Management Group, 2020).

Enterprise architecture
This describes the principles, practices, and parameters that guide the design and construction of applications and systems.

The Enterprise Architecture

The enterprise architecture concerns the strategic, tactical, and operational management of assets, resources, activities, and results. It guides organizational sustainability and growth, and strives to continually improve IT-business alignment (Gellweiler, 2020). In the enterprise architecture, the design (architecture) descriptions are divided into different layers (views); these frameworks are typically designed to fit the specific organization in which they are being implemented. However, such frameworks still contain a format or structure of generically applicable domains. For example, at the highest level, it starts with the business or enterprise domain that drives the (logical) design of the IT services required to enable the business. IT services are then enabled by applications, utilizing data and information, which, in turn, directs the design of the technological platforms (i.e., the physical IT assets and infrastructure). Each of these dictates the design and implementation of the one that follows; each layer also gives feedback to the previous layer in order to enable it. This high-level view is illustrated below.

Figure 49: A High-Level Enterprise Architecture Framework



Source: Venter, 2021.

Using a similar structure as that illustrated above, the TOGAF standard is an open-source enterprise architecture methodology and framework commonly applied to improve business efficiency (Josey, 2018). It guides organizations to effectively derive detailed processes and interactions of these layers based on business requirements.

TOGAF

TOGAF is developed and maintained by The Open Group, who work within the Architecture Forum (The Open Group, n.d.). The initial version was developed in 1995 and was based on the US Department of Defense Technical Architecture Framework for Information Management (TAFIM) (Josey, 2018). It is constantly evolving and being adapted to align with current market demands and new developments. The latest version, TOGAF 9.2, was published in 2020.

TOGAF is based on an iterative process model. Furthermore, it is modular and aligned with the Architecture Development Method (ADM). The ADM guides the methodical and cyclical development of an organization-specific and business-oriented enterprise architecture. It provides detailed descriptions of the approach, along with the relevant objectives and deliverables that aim to derive requirements for the TOGAF architecture types. TOGAF guides the architectural design in terms of architecture partitioning methods and provides a repository for architecture, as well as a framework for required capabilities, as follows (The Open Group, n.d.):

- Partitioning describes methods and techniques that facilitate effective partitioning of the different architectures.

- The architecture repository provides a logical information model to integrate and store all outputs that result from executing the ADM.
- The capability framework defines the organization, as well as relevant skills, roles, and responsibilities required to establish and maintain an operational and effective enterprise architecture capability.

TOGAF consists of an extensive reference library that includes various best practice “guidelines, templates, patterns, and other forms of reference material to accelerate the creation of new architectures for the enterprise” (The Open Group, n.d., Section 1.2, para. 1). It supports business architecture; data architecture; application architecture; and technology architecture (Josey, 2018, p. 5). They are defined as follows:

- Business architecture refers to the overall business strategy and defines the detailed structure and governance, as well as key business processes.
- Data architecture describes the organization of logical and physical data assets, as well as resources required to manage data and information.
- Application architecture provides a detailed plan (blueprint) used to deploy individual applications and describes how they interact with each other; it also explains how each application relates to the core business processes.
- Technology architecture relates to logical software and hardware capabilities, as required to deploy business, data, and application services.

IT Governance

IT (and business) governance are applied to effectively manage enterprise structures. IT governance is directed and guided by global standards, for example, ISO 21505:2017 and ISO/IEC 38500:2008. The Information Systems Audit and Control Association (ISACA®) defines IT governance as “the processes that ensure the effective and efficient use of IT in enabling an organization to achieve its goals” (Renard, 2016, p. 1). They divide IT governance processes into management of the demand side and management of the supply side, i.e., IT demand governance (ITDG) and IT supply-side governance (ITSG). ITDG includes the processes used to select, implement, and measure the business benefits of IT investments; it involves businesses dictating to IT what they should be focusing on, based on market demands and the needs of business, and is a business management responsibility. ITSG is the responsibility of IT management and leadership; it involves ensuring effective, efficient, and compliant IT operations (Gartner, 2020). Accordingly, IT governance is a combined concern of both business and IT. Effective governance requires that the IT strategy aligns with business strategy. COBIT is widely used as a framework for IT governance and management. It is applied to assess and improve IT processes and operations (including both the demand and supply side) and reduce the risks associated with unqualified use of IT (Dziak, 2019). In brief, COBIT provides a baseline framework and measurement criteria for IT.

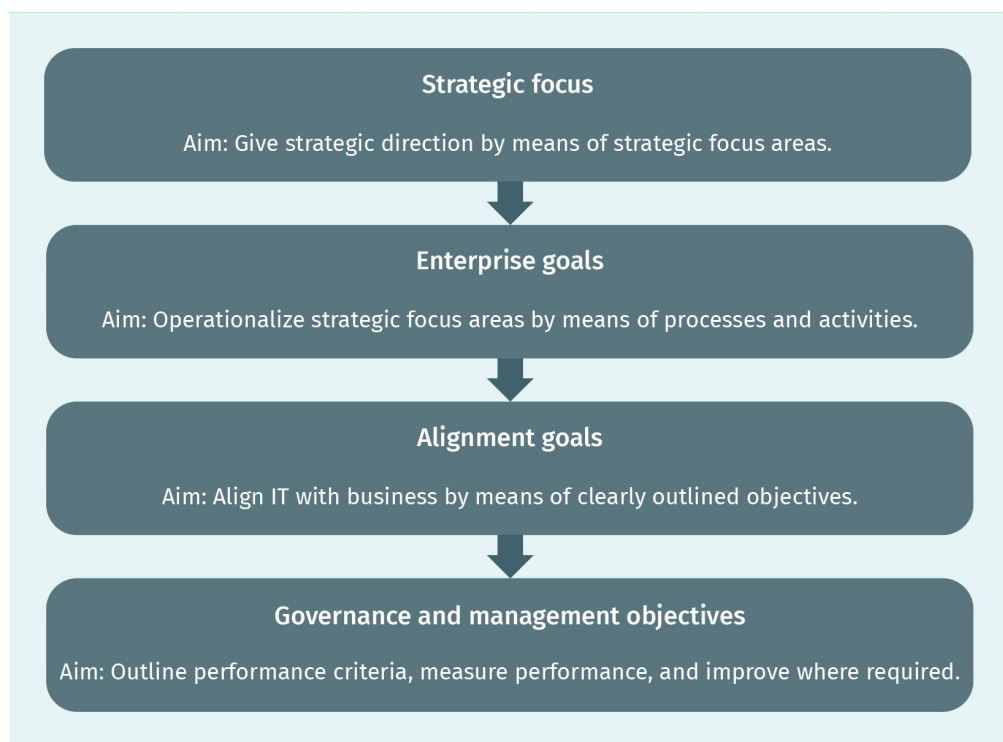
COBIT

COBIT was created in 1996 by ISACA for IT management and governance (ISACA, n.d.-a). The initial aim of COBIT was to normalize standards and uses of IT across different fields and industries and facilitate proper assessment (and audits) of IT systems (Dziak, 2019). It

has continued to evolve, and the latest version, COBIT 2019, is broad and all-inclusive. It is comprehensive and complicated in its entirety, but can be tailored to individual organizations, as per their requirements, to implement effective internal control measures (Rafeq, 2019). With the latest version, ISACA (n.d.-a) aims to facilitate the development of flexible and collaborative governance structures.

COBIT calls for an alternative approach to traditional functional management. Functional management focuses on managing individual functional areas, i.e., as silos, and process management focuses on the flow of related activities to achieve an objective. It links business and IT goals, measures responsibilities of both IT and business teams, and builds on the premise that IT and business must work together to achieve goals. So, when considering implementing a framework, such as COBIT, organizational strategic drivers must be considered first. These drive enterprise goals, which, in turn, shape IT-business alignment goals that are used to derive relevant (IT) governance and management objectives. This is illustrated below.

Figure 50: Organizational Strategic Focus Areas, Goals, and Objectives



Source: Jamsa & Harkiolakis, 2019.

Organizations have one primary, and possibly one additional (secondary), strategic driver, outlining the strategic focus areas. These are typically as follows (Cooke, 2020):

- Growth/acquisition (p. 8), i.e., the organization focuses on growing its revenue.
- Innovation/differentiation (p. 8), i.e., the organization wants to offer different and innovative products and services.
- Cost leadership (p. 8), i.e., they focus on minimizing cost in the short term.

- Client service/stability (p. 8), i.e., the organization aims to provide a stable, client-oriented service.

Following this strategy, relevant enterprise goals must be achieved. Enterprise goals include the processes and activities that aim to operationalize strategic focus areas. For example, they detail business processes to optimize products and services; processes used to innovate and grow; management of business risks; compliance with laws and regulations (as well as internal policies); quality information; management of costs and finances; and development and motivation of staff to ensure optimal productivity (Cooke, 2020). Furthermore, alignment goals emphasize the alignment of Information and Technology (I&T) with business. These are as follows (Cooke, 2020):

- I&T compliance and support for business compliance with external laws and regulations
- Managed I&T-related risk
- Realized benefits from I&T-enabled investments and services portfolios
- Quality of technology-related financial information
- Delivery of I&T services in line with business requirements
- Agility to turn business requirements into operational solutions
- Security of information, processing infrastructure and applications, and privacy
- Enabling and supporting business processes by integrating applications and technology
- Delivery of programs on time, on budget, and meeting requirements and quality standards
- Quality of I&T management information
- I&T compliance with internal policies
- Competent and motivated staff with mutual understanding of technology and business
- Knowledge, expertise, and initiatives for business innovation (p. 7)

Lastly, governance and management objectives are defined and measured. Criteria are explicitly defined in order to determine whether the performance of IT services and functions are on track or whether adjustments are needed. The core of COBIT 2019 contains governance objectives and management objectives. Each of these is described in terms of its purpose, how it ties in with the larger organization, and how it aligns with goals. They are to be applied as a baseline for tailored IT strategies and categorized according to relevant domains. For example, the COBIT governance objectives are positioned in the Evaluate, Direct, and Monitor (EDM) domain, which includes the following objectives (Edmead, 2020):

- EDM01: Ensured Governance Framework Setting and Maintenance
- EDM02: Ensured Benefits Delivery
- EDM03: Ensured Risk Realization
- EDM04: Ensured Resource Optimization
- EDM05: Ensured Stakeholder Engagement (p. 1)

The COBIT management objectives are categorized in the following domains (Edmead, 2020):

- APO Align, Plan and Organize
- BAI Build, Acquire and Implement
- DSS Deliver, Service and Support

- MEA Monitor, Evaluate and Assess (p. 2)

COBIT 2019 also effectively integrates with other frameworks. For example, it is compatible with ISO/IEC, TOGAF, and ITIL. It also aligns well with the CMMI capability and maturity framework, as it proposes a COBIT Performance Management (CPM) model to apply to score processes on a quantitative scale of 0–5 (ISACA, n.d.-a). With COBIT 2019, ISACA (n.d.-a) moved to an open-source model and welcomes feedback from practitioners to continuously improve it.

Agile and Software Development Governance

Agile (or lean) governance is a relatively new term that is being used increasingly often. It is currently being explored by both academics and practitioners. It involves investigating Agile and lean practices to govern, for example, software engineering (Luna et al., 2014). However, there are still some concerns and ongoing debate on this topic (Juiz & Colomo-Palacios, 2020). For now, Software Development Governance (SDG) remains an emerging subject area that aims to empower software teams in reaching project goals. Broadly speaking, it entails establishing and defining relevant roles, responsibilities, and associated decision-making authorities, as well as engaging in regular reflection to assess processes and products. The following high-level steps have been proposed for iteration (Dubinsky & Kruchten, 2009):

- Set goals and assign roles and decision rights, and the responsible individuals can then reach set goals
- Determine measurements and policies, which facilitates understanding of and control over behavior and organizational performance
- Implement the above mechanisms practically
- Assess implementation of the above and refine and evolve goals

6.2 Process Design and Deployment

Software processes, as well as governance processes and systems, must be suited to the area of application. Hence, relevant design factors must be considered to ensure alignment with business goals and objectives, the organizational context, strategy, and tactical implementation choices. Following process design, processes are deployed and implemented.

Process Design Factors

Process design factors that relate to the organizational context refer to external influences, e.g., the organization's geopolitical situation, country-specific economic policies, or regulations. Strategic factors include, for example, the business strategy. It includes the role that IT plays within the organization (whether it is perceived as a business differentiator, or is merely supportive of and enables the business), and its appetite for (or aversion to) risk. Tactical implementation choices include the practical decisions regarding resourcing models (e.g., outsourcing versus insourcing versus a blended approach, the use of cloud

computing, purchasing or leasing of IT assets, or approaches to standardization); IT and software models and processes (e.g., Agile versus traditional versus blended development approaches, and the adoption and use of lean and DevOps practices); and choices regarding technology adoption (do they choose to be first adopters of leading and innovate technology, or be the late majority?) (Rafeq, 2019). As previously discussed, IT processes, tools, and roles are most effectively established within a framework, such as ITIL.

Process Deployment

Process deployment entails implementation of a process model, i.e., bridging the theory-practice gap. Process deployment is challenging, expensive, and may even be met with apathy and resistance. In order to effectively deploy processes, it is essential to clearly express expectations, enable and verify the implementation of processes, and reward successful implementation (Kneuper, 2018). The deployment of processes results in an organizational change, which must be managed appropriately. Governance models, such as COBIT, and measurement models, such as CMMI models, are applied in this regard.

6.3 Process Tailoring

Software processes define and describe the relevant practices and approaches to follow and are useful for standardization. However, ever-changing market demands, as well as evolving customer requirements and technological developments, necessitate that processes be tailored to specific projects and goals. **Process tailoring** is the modification or adaptation of a standard process to fit the organization and the specific circumstances of a project. It can involve adding supplementary segments, modifying parts, or even removing portions of the process. Agile methods are inherently tailored and tailorable by nature. Plan-driven methods are extremely comprehensive, but can also be “tailored down” (Boehm & Turner, 2004, pp. 36–37); unfortunately, the “tailorable methods are used—and verbally abused—by developers and acquirers alike” (pp. 36–37), much to the frustration of the “expert methodologists” (pp. 36–37) that provide extensive guidelines and examples for the tailoring of the processes.

Process tailoring
This is the modification or adaptation of a standard process to fit the organization and circumstances of a project.

Process tailoring is challenging; ascertaining the best, most applicable version of a process for the specific circumstances beforehand is difficult. Caution must be taken to ensure that the tailoring of a process does not compromise the integrity of its core, and careful consideration is required to ensure that the consistency of the original process model is preserved (Kneuper, 2018). Consequently, tailoring guidelines are defined to guide process adaptations. They can include the extent and levels of tailoring that are allowed, guidelines regarding formal documentation of tailoring aspects, and steps to follow when tailoring a process. Guidelines, such as the ones published by IEEE (1998), aid the tailoring of processes for specific business domains, but are not sufficient for tailoring processes for specific projects. Process tailoring can also be automated, as in the case of the V-Modell XT Assistant, which allows tailoring according to a set of pre-defined criteria. Xu and Ramesh (2008) suggest the following steps to tailor software processes (plans for tailoring can often not be finalized prior to commencement of a project, so these steps are performed iteratively):

- Evaluate project goals and environment. This involves evaluation of the product and process goals and consideration of the specific characteristics of the project, team, stakeholders, and organization in order to ensure consistency.
- Assess challenges. Challenges typically arise related to resources, communication, requirements management, political issues, and technical challenges.
- Determine tailoring strategies for the various process elements. This refers to decisions regarding inclusion or exclusion, altering of tasks and artifacts, roles, and sequencing, as well as iterating work.
- Tailor software processes. This involves using process validation to ensure consistency with goals, the environment, the execution, and its subsequent evaluation.

6.4 Process Assessment, Improvement, and Measurement

Processes and process models must be assessed to evaluate their quality. In addition, performance and effectiveness should be assessed, and processes must be amended and improved when required. Assessments should aim to gauge the expectations of customers and stakeholders and determine whether processes deliver reliable, cost effective products and services that adhere to set quality criteria. Performance management planning entails the setting of relevant objectives, goals, and expectations, and performance measurement processes should provide direction for tasks and activities, measure their performance, and compare the results against set objectives (IT Governance Institute, 2005).

Software Process Assessment and Improvement

Software Process Improvement (SPI) aims to improve software processes and apply various standards and methods to assess the quality and maturity of processes. *The SPI Manifesto* originated at the EuroSPI Conference in Spain in 2009. It defines values and principles to achieve SPI (EuroAsiaSPI², n.d., Values section, Principles section):

- In terms of people, SPI “must involve people actively and affect their daily activities” (Values section), namely
 - know the culture and focus on needs,
 - motivate all people involved,
 - base improvement on experience and measurements, and
 - create a learning organization (Principles section).
- In terms of the business, SPI “is what you do to make business successful” (Values section), namely
 - support the organization’s vision and objectives,
 - use dynamic and adaptable models as needed, and
 - apply risk management (Principles section).

- In terms of change, SPI “is inherently linked with change” (Values section), and should
 - manage the organizational change in your improvement effort,
 - ensure all parties understand and agree on process, and
 - do not lose focus (Principles section).

Quality management (QM) is key aspect of process improvement. QM is founded on the plan-do-check-act (PDCA) cycle, based on Walter A. Shewhart’s work on controlling the quality of products (Shewhart, 1931). PDCA involves planning (identifying what to improve upon, and how to achieve improvement); taking action to implement the planned improvement actions; checking the outcome (and success) of the implementation (i.e., confirming whether the problems were resolved); and acting accordingly to adjust and implement more improvement actions if needed.

Process Capability and Maturity Measurement

The relative capability and maturity of the processes of an organization can be effectively appraised using **capability maturity models**, such as those provided by the CMMI Institute. These maturity models were initially created so that the US Department of Defense could evaluate the quality and capability of software contractors (ISACA, n.d.-c). At present, they are applied in the software engineering domain and beyond. They serve several purposes, including demonstrating organizational capability to external stakeholders by illustrating how processes compare to best practices and identifying areas of improvement, assisting organizations to meet their contractual obligations, and supporting corporate and IT governance.

Capability maturity models

These are models used to assess the relative capability and maturity of organizational processes.

The defined maturity levels refer to the maturity of processes within an organization. They offer a phased means of appraising and improving processes, subsequently improving performance. Process maturity is, according to the CMMI suite of models (n.d.-b), described using a quantitative scale which ranges from a maturity level of zero to a maturity level of five. Maturity levels are evolutionary, meaning that one level must be fulfilled before moving to the next (ISACA, n.d.-b). These levels define whether a process is incomplete, initial, managed, defined, quantitatively managed, or optimizing. The maturity levels are defined as follows (ISACA, n.d.-b):

- Incomplete. Work is done ad-hoc, in a way that is undefined and unclear, and the completion of work cannot be guaranteed.
- Initial. Work is completed in an unpredictable, reactive way, and budgets and schedules are often exceeded.
- Managed. Project management principles are applied to pro-actively plan, organize, and finish work in a controlled manner.
- Defined. Work is planned and executed using projects positioned within organizationally-driven programs and portfolios. These are guided by organization-wide standards.
- Quantitatively managed. Work is effectively measured and controlled by means of quantitative objectives that are predictable and data-driven in order to meet all the stakeholders’ needs.
- Optimizing. The organization is stable, but flexible: it is responsive, Agile, innovative, and strives to improve continuously.

Capability levels describe the performance and process improvement achievements of organizational processes in specific practice areas. The scale applied to indicate capability consists of levels zero to three. The levels are incomplete, initial, managed, or defined. It is important to note that capability level subsumes the practices of level 1, while level 3 builds on the practices of level 2 (ISACA, n.d.-b). The capability levels are defined as follows (ISACA, n.d.-b, Capability Levels section):

- “Incomplete” aims to address inconsistent performance, indicates that an approach is incomplete, and implies that the intent of a practice may not be met (para. 2).
- “Initial” aims to address performance issues and indicates that practices are still incomplete and will not meet the full intent of a practice area (para 3.).
- “Managed” aims to identify and monitor progress towards meeting project performance objectives and indicates that practices are simple, but complete enough to address the practice area’s full intent (para. 4).
- “Defined” aims to achieve performance objects in the project as well as the organizational sphere, and indicates the use of organizational standards and process tailoring (para 5.).

6.5 Tool Support

Due to the complexity of software processes, suitable tools are employed to define, develop, manage, and enact processes. In line with the distinction made by Kneuper (2018), we will look at these in the two following groups: tools that support process modeling and management, and tools that support process enactment.

Process Modeling and Management

Process editors are used to document processes using relevant modeling notations. Standard text XML editors can be used to document processes. However, use of complex meta-models, e.g., the Software Process Engineering Metamodel (SPEM) and the V-Modell XT, can be simplified with particularly tailored editors. Custom-made editors are used to ease process tailoring. For example, the Eclipse Process Framework (EPF) composer can be used to define, adapt, and enact SPEM-based software processes, whereas two different tools, e.g., the V-Modell XT Editor and V-Modell XT Assistant, can be used to edit and tailor processes that apply the V-Modell XT.

Advanced editors facilitate the generation of syntactically correct process definitions and map and indicate applicable interrelationships and consistency conditions accurately (Kneuper, 2018). Process visualization tools are useful to visualize processes in their entirety or filter out detail to only show parts of processes, as applicable to different team members (Kneuper, 2018).

Process Enactment

Process enactment tools are applicable to all stages of a software product life cycle. Computer-based tools that support software development increase the development efficiency, quality, and maintainability of the software. This supports the documentation of processes, simplifies management of projects, and improves collaboration within and among teams (Kneuper, 2018). These tools can also be useful to facilitate coordination and consistent version control in large and complex projects (Avison & Fitzgerald, 2006).

Though projects are executed by different teams with different individual aims, they still work towards a shared goal. Integrated tools, e.g., in Integrated Development Environments (IDE), support tasks such as code editing, design, compiling and debugging, source code control, and build management (Kneuper, 2018). However, IDE services must still be used thoughtfully. For example, developers are often unaware of all the methods that can be invoked on given variables and therefore appreciate the concept of intelligence code completion. However, some variables can result in hundreds of method proposals, and choosing the correct one is overwhelming (Bruch et al., 2010).

Commercial off-the-shelf (COTS) software often requires appropriate installation support tools. Furthermore, Agile practices, such as DevOps, call for automated software integration, deployment, and delivery, and therefore necessitate the use of applicable tools. These are very expensive to procure and complex to deploy. For example, Infrastructure as Code (IaC) is an approach that supports code-centric tools, whereas model-driven tools can also be used (Sandobalin et al., 2020). The advantages and disadvantages of these tools, as well as the organizational context where they are applied, determine their usefulness.



SUMMARY

Process management and governance extends beyond software engineering; all IT assets and services should be managed holistically to add value. Various process models, development approaches, and management frameworks have been introduced throughout this course. However, in the end, we should acknowledge that models, approaches, and frameworks cannot be implemented in isolation. Processes should be managed and governed within the framework of an enterprise architecture. Overarching management and governance frameworks should be complemented by responsive Agile processes and similar work approaches, and vice versa. Accordingly, processes should be designed and deployed to fit the specific organization, and business objectives must always drive process design and implementation. Standardized processes can be tailored to fit organizations, specific projects, and teams. However, tailoring should be done in such a way that process consistency and integrity remain intact.

Processes, within all practice areas, can be effectively appraised using the levels shown in capability and maturity models. This can provide an organization with a high-level roadmap for improvement, i.e., improvement actions to be implemented in order to systematically improve operations. Additionally, supporting tools go a long way to simplify complex models and processes. However, the selection, implementation, and application of supporting tools should still be performed appropriately and in the context of the business and organizational culture.