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 Iivari 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 ensurance 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

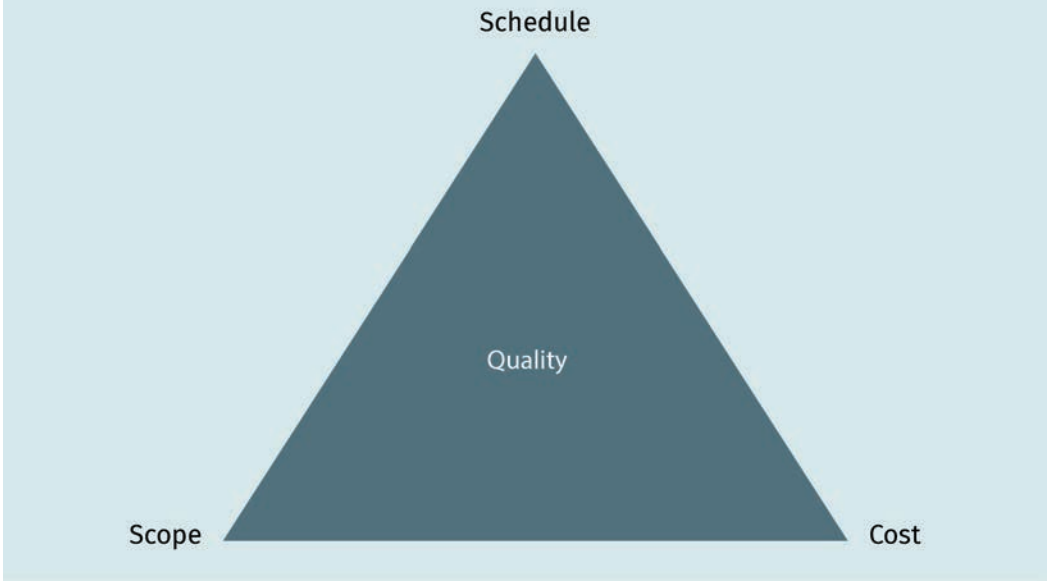
manner, and facilitate the development of software that is consistent with quality criteria, as well as within time and budget constraints. Successfully executing such projects requires the rigorous application of a suitable approach and appropriately managing the associated risks and challenges. However, 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.

The Project Triangle—Indicating Project Constraints



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 electrons 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.

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

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 Bemer, 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

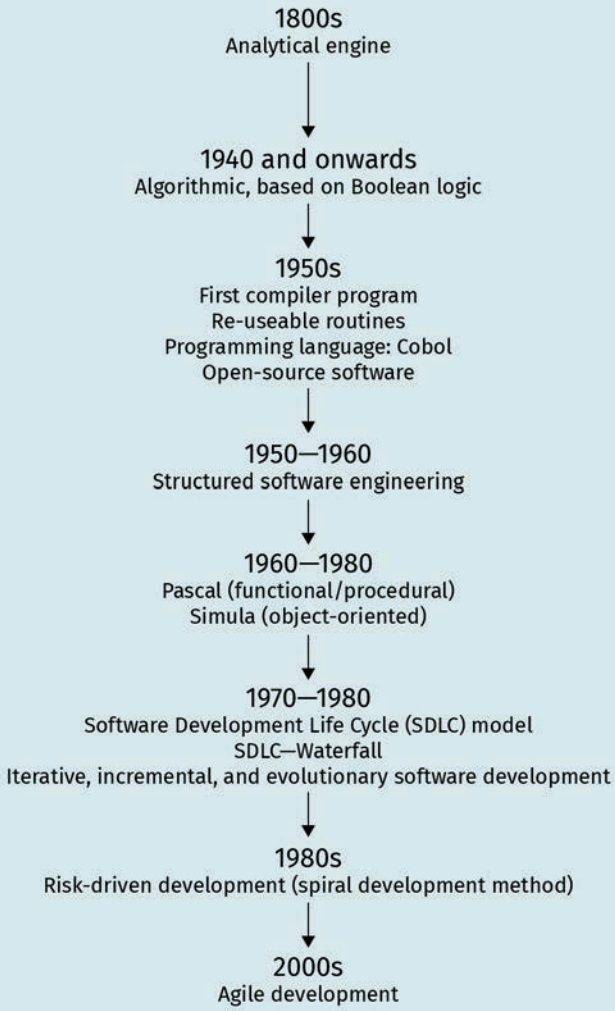
The software development life cycle (SDLC) model, as introduced by Royce (1970), is still viewed by many as the first and traditional software development approach as it greatly influenced software development practices (Avison & Fitzgerald, 2006). The SDLC model was, however, more generally termed, 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 evolvement 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.

High Level Historical Overview



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 pha-

ses. These involve a feasibility study, requirements elicitation, 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.

Knowledge Check

Did you understand this unit?

You can check your understanding by completing the questions for this unit on the learning platform.

Good luck!