

INTRODUCTION

ACRONYMS

KA	Knowledge Area
SWEBOK	Software Engineering Body of Knowledge

Publication of the 2014 version of the *Guide to the Software Engineering Body of Knowledge (SWEBOK Guide V3)* was a major milestone in establishing software engineering as a recognized engineering discipline. The goal of developing this update (Version 4) to the *SWEBOK Guide* is to improve the *Guide's* currency, readability, consistency and usability. The content of the *Guide* consists of 18 knowledge areas (KAs) followed by several appendixes. A KA is an identified area of software engineering defined by its knowledge requirements and described in terms of its component processes, practices, inputs, outputs, tools and techniques. Three appendixes provide, respectively, the specifications for the KA descriptions, an annotated set of relevant standards for each KA, and a list of references cited in the *Guide*.

All KAs have been updated to reflect changes in software engineering since the publication of the *Guide V3*, including modern development practices, new techniques, and the advancement of standards. One significant change is that Agile and DevOps have been incorporated into almost all KAs because these models have been widely accepted since the previous publication of the *Guide*. Agile models typically involve frequent demonstrations of working software to a customer in short, iterative cycles. Agile practices exist across KAs. Furthermore, emerging platforms and technologies,

including artificial intelligence (AI), machine learning (ML) and the Internet of Things (IoT), have been incorporated into the foundation KAs.

To reflect areas that are becoming particularly important in modern software engineering, the following KAs have been added: the Software Architecture KA, Software Security KA and Software Engineering Operations KA.

This *Guide*, written under the auspices of the Professional and Educational Activities Board of the IEEE Computer Society, represents a next step in the evolution of the software engineering profession.

WHAT IS SOFTWARE ENGINEERING?

IEEE Std. 610.12-1990 Glossary of Software Engineering Terminology and ISO/IEC/IEEE Systems and Software Engineering Vocabulary (SEVOCAB) defines software engineering as “the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.”¹ Historically, software engineering has been defined in various ways, such as “the practical application of scientific knowledge to the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them” [1] and “the technological and managerial discipline concerned with systematic production and maintenance of software products that are developed and modified on time and within cost estimates” [2]. Although these definitions differ in detail, they have an

¹ http://pascal.computer.org/sev_display/index.action.

essential commonality in that they both deal with software development and maintenance. Furthermore, the application of scientific knowledge (mentioned in the first definition) can be described as a technological discipline (a phrase used in the second definition). As “scientific” implies a systematic and quantifiable approach, the initial definition also expresses an idea common in past definitions of the discipline.

Software engineering occupies a position between the mathematical and physical disciplines of computer science and technology on the one hand and the work of applying those findings to solve the problems of particular application domains on the other [2]. Science is about discovering new things. On the other hand, engineering is about applying that knowledge to solve real-world problems cost-effectively. As such, the engineering discipline of a given scientific field requires skills and knowledge about relevant “practice.” Further, as engineering concerns cost-effective solutions to real-world problems, all engineering disciplines involve engineering economics, which is the analysis of theoretically possible solutions to identify the most cost-effective one. In essence, this *Guide* distills the relevant theory of computer science into the two foundation KAs, while the remaining KAs catalog the practice and engineering economics of software engineering.

Software engineering techniques can be viewed as specializations of techniques of more general disciplines, such as project management, systems engineering and quality management [2]. Furthermore, a software project must implement requirements imposed by cross-cutting disciplines such as dependability and safety.

Software engineering and computer science are related but distinct in the same way chemical engineering and chemistry are related but distinct. Scientific disciplines, such as computer science and chemistry, aim

to extend human knowledge. Effective requirements elicitation techniques, design principles like cohesion and coupling, appropriate branch-merge strategies, conducting a proper peer review, and assessing the cost of quality are a few examples of critical software engineering practices that are of little or no concern to computer science. In engineering, science and practice are applied to generate potential solutions to the real-world problem, and engineering economics is used to identify the most cost-effective one. In the same way that it would not make sense to send a chemist to solve a chemical engineering problem, it does not make sense to send a computer scientist to solve a software engineering problem.

In addition to computer science, software engineering is related to several other disciplines and professional areas, such as project management, quality management, industrial engineering, dependability engineering, and safety and security engineering.

WHAT ARE THE OBJECTIVES OF THE SWEBOK GUIDE?

The *Guide* should not be confused with the body of knowledge itself, which exists in the published literature. The *Guide*’s purpose is to describe the generally accepted portion of the body of knowledge, organize that portion, and provide topical access to it.

The *SWEBOk Guide* was established with the following five objectives:

1. To promote a consistent view of software engineering worldwide
2. To specify the scope and clarify the place of software engineering with respect to other disciplines, such as computer science, project management, computer engineering and mathematics
3. To characterize the contents of the software engineering discipline

4. To provide topical access to the Software Engineering Body of Knowledge
5. To provide a foundation for curriculum development and for individual certification and licensing materials

The first objective, to promote a consistent worldwide view of software engineering, was supported by a development process that engaged about NNNN reviewers from NNNN countries. More information regarding the development process can be found at www.swebok.org. Professional and learned societies and public agencies involved in software engineering were contacted, made aware of this project to update the *SWEBOK Guide*, and invited to participate in the review process. Associate editors were recruited from North America, the Pacific Rim Europe, and Asia. Presentations on the project were made at various international venues.

The second objective, to specify the scope of software engineering, underlies the fundamental organization of the *Guide*. Material that falls within this discipline is organized into the 17 KAs listed in Table I.1. Each KA is treated as a chapter in this *Guide*.

Table I.1. The 17 SWEBOK KAs

1. Software Requirements
2. Software Architecture
3. Software Design
4. Software Construction
5. Software Testing
6. Software Engineering Operations
7. Software Maintenance
8. Software Configuration Management
9. Software Engineering Management
10. Software Engineering Models and Methods
11. Software Engineering Process
12. Software Quality

13. Software Security
14. Software Engineering Economics
15. Software Engineering Professional Practice
16. Computing Foundations
17. Mathematical Foundations
18. Engineering Foundations

In specifying the scope of the discipline, it is also important to identify disciplines that intersect with software engineering. To this end, the *SWEBOK V4 Guide* continues to recognize seven related disciplines, listed in Table I.2. Software engineers should, of course, be knowledgeable about these disciplines (and KA descriptions in this *Guide* might refer to them). However, characterizing the knowledge of related disciplines is not an objective of the *SWEBOK Guide*.

Table I.2. Related disciplines

1. Computer engineering
2. Computer science
3. General management
4. Mathematics
5. Project management
6. Quality management
7. Systems engineering

The relevant elements of computer science and mathematics are presented in the Computing Foundations KA and Mathematical and Engineering Foundations KAs of the *Guide* (Chapters 16 and 17).

HIERARCHICAL ORGANIZATION

The organization of the KA chapters supports the third project objective — to characterize the contents of software engineering. The detailed specifications provided by the project's editorial team to the associate editors regarding the contents of the KA descriptions can be found in Appendix A.

The *Guide* uses a hierarchical organizational structure to decompose each KA into a set of topics with recognizable labels. Each KA provides a two- or three-level breakdown, which provides a reasonable way to find topics of interest. The *Guide* treats the selected topics in a way that is compatible with major schools of thought and separates the topics into subtopics that are generally found in industry and in software engineering literature and standards. The breakdowns are not designed for particular application domains, business uses, management philosophies, development methods and so forth. Each topic description is meant only to give the reader a general understanding of the topic and to enable the reader to find reference material. The body of knowledge is found in the reference materials, not in the *Guide*.

Software plays a core role in various application and technological domains, such as automotive, legal, health care, and finance. Differences in application domains and business models (e.g., custom applications, and open source applications) and system types (e.g., enterprise and cloud systems, embedded and IoT systems, and AI/ML-based systems) may influence what practices are adopted. Major special techniques and practices specific to certain system types are also discussed in some KAs, especially the Software Requirements KA, the Software Testing KA, the Software Quality KA, the Software Security KA and the Computing Foundations KA.

REFERENCE MATERIAL AND MATRIX

To provide topical access to the knowledge — the fourth project objective — the *Guide* identifies authoritative reference material for each KA. In addition, Appendix C provides a Consolidated Reference List for the entire *Guide*. Each KA includes relevant references

from the Consolidated Reference List as well as a matrix connecting the reference materials to the topics covered.

Please note that the *Guide* does not attempt to be comprehensive in its citations. Much suitable and excellent material is not referenced. However, the material included in the Consolidated Reference List provides further information about the topics described.

DEPTH OF TREATMENT

To achieve the *Guide*'s fifth objective — to provide a foundation for curriculum development, certification and licensing — the criterion of *generally accepted* knowledge has been applied. This is distinct from advanced and research knowledge (on the grounds of maturity) and from specialized knowledge (on the grounds of generality of application).

The equivalent term *generally recognized* comes from the Project Management Institute:²

“Generally recognized means the knowledge and practices described are applicable to most projects most of the time, and there is consensus about their value and usefulness.”

However, the terms *generally accepted* and *generally recognized* do not imply that the designated knowledge should be uniformly applied to all software engineering endeavors — each project's needs determine what knowledge to apply, and how. However, competent, capable software engineers should be equipped with this knowledge for potential application. Therefore, generally accepted knowledge should be included in the study material for the software engineering licensing examination that graduates take after gaining four years of work experience. Although this criterion is

² *A Guide to the Project Management Body of Knowledge*, 5th ed.,

specific to the US style of education and does not necessarily apply to other countries, we deem it useful.

STRUCTURE OF THE KA DESCRIPTIONS

Each chapter provides a description of one of the KAs. These descriptions are structured as follows.

The introduction briefly defines the KA and presents an overview of its scope and its relationship with other KAs.

The breakdown of topics in each KA constitutes the core of the KA description, showing the decomposition of the KA into subareas, topics and subtopics. For each topic or sub-topic, a short description is given, along with one or more references.

These reference materials were selected as the best available presentation of knowledge related to the topic. A matrix links the topics to the reference materials.

The last part of each KA description is the list of recommended references and suggested further reading. Relevant standards for each KA are presented in Appendix B of the *Guide*.

APPENDIX A. KA DESCRIPTION SPECIFICATIONS

Appendix A describes the specifications provided by the editorial team to the associate editors for the content, recommended references, format and style of the KA descriptions.

APPENDIX B. ALLOCATION OF STANDARDS TO KAs

Appendix B presents an annotated list of the relevant standards, mostly from the IEEE and the ISO, for each of the *SWEBOK Guide*'s KAs.

APPENDIX C. CONSOLIDATED REFERENCE LIST

Appendix C contains the consolidated list of recommended references cited in the KAs. (These references are marked with an asterisk (*) in the text.)

REFERENCES

- [1] Barry W. Boehm, "Software Engineering," IEEE Transactions on Computers, Vol. C-25, No. 12, 1976.
- [2] James W. Moore, "Software Engineering Standards: A User's Road Map," IEEE Computer Society, 1998.

CHAPTER 1

SOFTWARE REQUIREMENTS

ACRONYMS

ATDD	Acceptance Test Driven Development
BDD	Behavior Driven Development
CIA	Confidentiality, Integrity, and Availability
FSM	Functional Size Measurement
INCOSE	International Council on Systems Engineering
JAD	Joint Application Development
JRP	Joint Requirements Planning
RUP	Rational Unified Process
SME	Subject Matter Expert
SysML	Systems Modeling Language
TDD	Test Driven Development
UML	Unified Modeling Language

If a team does a poor job of determining the requirements, the project, the product or both are likely to suffer from added costs, delays, cancellations and defects. One reason is that each software product requirement generally leads to many design decisions. Each design decision generally leads to many code-level decisions. Each decision can involve several test decisions, as well. In other words, determining the requirements correctly is high-stakes work. If not detected and repaired early, missing, misinterpreted and incorrect requirements can induce exponentially cascading rework to correct them.

Real-world software projects tend to suffer from two primary requirements-related problems:

1. incompleteness: stakeholder requirements exist that are not revealed and communicated to the software engineers;
2. ambiguity: requirements are communicated in a way that is open to multiple interpretations, with only one possible interpretation being correct.

INTRODUCTION

Software requirements should be viewed from two perspectives. The first is as an expression of the needs and constraints on a software product or project that contribute to the solution of a real-world problem. The second is that of the activities necessary to develop and maintain the requirements for a software product and for the project that constructs it. Both perspectives are presented in this knowledge area (KA).

Beyond the obvious short-term role requirements play in initial software construction, they also play a less recognized but still important role in long-term maintenance. Upon receiving software without any supporting documentation, a software engineer has several means to determine what that code does, such as execute it, step through it with a debugger, hand-execute it, statically analyze it, and so on. The challenge is determining what that code is *intended to do*. What is generally referred to as a

bug — but is better called a *defect* — is simply an observable difference between what the software is intended to do and what it does. The role of requirements documentation throughout the service life of the software is to capture and communicate intent for software engineers who maintain the code but might not have been its original authors.

The Software Requirements KA concerns developing software requirements and managing those requirements over the software's service life. This KA provides an understanding that software requirements:

- are not necessarily a discrete front-end activity of the software development life cycle but rather a process initiated at a project's beginning that often continues to be refined throughout the software's entire service life;
- need to be tailored to the organization and project context.

The term *requirements engineering* is often used to denote the systematic handling of requirements. For consistency, the term *engineering* will not be used in this KA other than for software engineering per se.

The Software Requirements KA is most closely related to the Software Architecture, Software Design, Software Construction, Software Testing, and Software Maintenance KAs, as well as to the models topic in the Software Engineering Models and Methods KA, in that there can be high value in specifying requirements in model form.

This KA is also related to the Software Life Cycles topic in the Software Engineering Process KA, in that this KA's focus is on *what* and *how* requirements work can and should be done, whereas the project's life cycle determines *when* that work is done. For example, in a waterfall life cycle, all requirements work is essentially done in a discrete *Requirements phase* and is expected to be substantially complete before any

architecture, design and construction work occurs in subsequent phases. Under some iterative life cycles, initial, high-level requirements work is done during an *Inception phase*, and further detailing is done during one or more *Elaboration phases*. In an Agile life cycle, requirements work is done incrementally, just in time, as each additional element of functionality is constructed.

The *whats* and *hows* of software requirements work on a project should be determined by the nature of the software constructed, not by the life cycle under which it is constructed. Insofar as requirements documentation captures and communicates the software's intent, downstream maintainers should not be able to discern the life cycle used in earlier development from the form of those requirements alone.

This KA is also related, but somewhat less so, to the Software Configuration Management, Software Engineering Management and Software Quality KAs. Software CM approaches can be applied to trace and manage requirements; software quality looks at how well formed the requirements are, and engineering management can use the status of requirements to evaluate the completion of the project.

BREAKDOWN OF TOPICS FOR SOFTWARE REQUIREMENTS

The topic breakdown for the Software Requirements KA is shown in Figure 1.1.

1. Software Requirements Fundamentals

1.1. Definition of a Software Requirement [1*, c1pp5-6] [2*, c4p102]

Formally, a *software requirement* has been defined as [28]:

- a condition or capability needed by a user to solve a problem or achieve an objective;

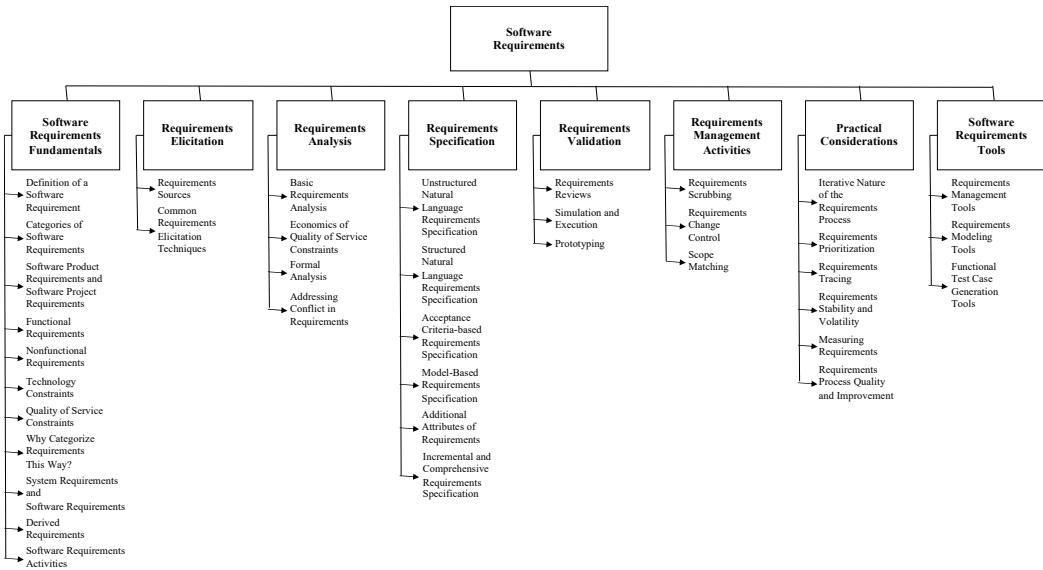


Figure 1.1. Breakdown of Topics for the Software Requirements KA

- a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification or other formally imposed document;
- a documented representation or capability as in (1) or (2) above.

This formal definition is extended in this KA to include expressions of a software project's needs and constraints.

At its most basic, a software requirement is a property that must be exhibited to solve a real-world problem. It might aim to automate all or part of a task supporting an organization's business policies and processes, correct existing software's shortcomings, or control a device — just a few of the many problems for which software solutions are possible.

Business policies and processes, as well as device functions, are often very complex. By extension, software requirements are often a complex combination of requirements from various stakeholders at different organizational levels who

are involved or connected with some aspect of the environment in which the software will operate.

Clients, customers and users usually impose requirements. However, other third parties, like regulatory authorities and, in some cases, the software organization or the project itself, might also impose requirements. (See also [5, c1] [6, c1] [9, c4].)

1.2. Categories of Software Requirements [1*, c1pp7-12] [2*, s4.1]

Figure 1.2 shows the categories of software requirements defined in this KA and the relationships among those categories. (See also [5, c1] [6, c1] [9, c4].) Each category is further described below.

1.3. Software Product Requirements and Software Project Requirements [1*, c1pp14-15]

Software product requirements specify the software's expected form, fit or function. *Software project requirements* — also called *process requirements* or, sometimes *business requirements* — constrain the project that constructs the software. Project requirements often constrain cost, schedule and/or staffing but can also constrain other aspects of a software project, such as testing

environments, data migration, user training, and maintenance. Software project requirements can be captured in a project charter or other high-level project initiation document. They are most relevant to how the project is managed (see the Software Engineering Management KA) or what life cycle process should be used (see the Software Engineering Process KA). This KA does not discuss software project requirements further.

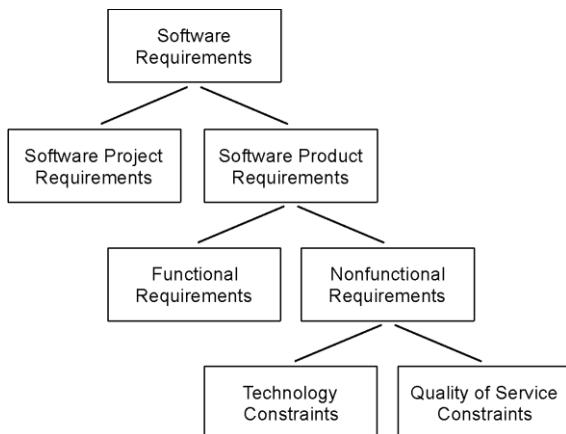


Figure 1.2. Categories of Software Requirements

1.4. Functional Requirements [1*, c1p9] [2*, s4.1.1]

Functional requirements specify observable behaviors that the software is to provide — policies to be enforced and processes to be carried out. Example policies in banking software might be “an account shall always have at least one customer as its owner,” and “the balance of an account shall never be negative.” Example processes could specify the meanings of depositing money into an account, withdrawing money from an account and transferring money from one account to another.

Even highly technical (nonbusiness-oriented) software, such as software that implements the Transmission Control Protocol/Internet Protocol (TCP/IP) network communications protocol, has policies and processes: “a Port shall be able to exist with zero, one, or many associated Connections, but a Connection shall exist on exactly one associated Port,” “acceptable states of a Connection shall be

‘listen,’ ‘syn sent,’ ‘established,’ ‘closing,’ . . . ,” and “if the time-to-live of a Segment reaches zero, that Segment shall be deleted.” (See [5, c1] [6, c10] [9, c4].)

1.5. Nonfunctional Requirements [1*, c1pp10-11] [2*, s4.1.2]

Nonfunctional requirements in some way constrain the technologies to be used in the implementation: What computing platform(s)? What database engine(s)? How accurate do results need to be? How quickly must results be presented? How many records of a certain type need to be stored? Some nonfunctional requirements might relate to the operation of the software. (See the Operation and Maintenance KA.) (See also [5, c1] [6, c11] [9, c4].)

The nonfunctional requirements can be further divided into technology constraints and quality of service constraints.

1.6. Technology Constraints

These requirements mandate — or prohibit — use of specific, named automation technologies or defined infrastructures. Examples are requirements to use specific computing platforms (e.g., Windows™, MacOSTM, Android OSTM, iOSTM), programming languages (e.g., Java, C++, C#, Python), compatibility with specific web browsers (e.g., Chrome™, Safari™, Edge™), given database engines (e.g., Oracle™, SQL Server™, MySQL™), and general technologies (e.g., Reduced Instruction Set Computer (RISC), Relational Database. A requirement prohibiting use of pointers would be another example. (See also [9, c4].)

1.7. Quality of Service Constraints

These requirements do not constrain the use of specific, named technologies. Instead, these specify acceptable performance levels an automated solution must exhibit. Examples are response time, throughput, accuracy, reliability and scalability. ISO/IEC 25010: “System and software engineering – Systems and software Quality Requirements and

Evaluation (SQuaRE) – System and software quality models” [27] contains a large list of the kinds of service qualities that can be relevant for software. (See also [9, c4].) Security is also a particularly important topic where requirements tend to be overlooked. (See the Security KA for details on the kinds of specific security requirements that should be considered.) (See also [2*, c13].)

1.8. Why Categorize Requirements This Way?

Categorizing requirements this way is useful for the following reasons:

- requirements in one category tend to come from different sources than other categories;
- elicitation techniques often vary by source;
- analysis techniques vary by category;
- specification techniques vary by category;
- validation authorities vary by category;
- the different categories affect the resulting software in different ways.

In addition, organizing the requirements in these categories is beneficial in the following ways:

- complexity can be better managed because different areas can be addressed separately; software engineers can deal with policy and process complexities without worrying about automation technology issues at the same time (and vice versa). One large problem becomes two smaller ones. This is classic *divide and conquer* complexity management;
- distinct areas of expertise can be isolated; stakeholders, not software engineers, are the experts in the policies and processes to be automated. Software engineers, not stakeholders, are the technology experts. When a business expert is given interspersed functional and nonfunctional requirements for review or validation, they might give up because they don’t understand — or even care about — the technology issues. The relevant requirements

reviewer can focus on just the subset of requirements relevant to them.

The *Perfect Technology Filter* originally described in [18, c1-4] but also explained in [8] and [9, c4] helps separate functional from nonfunctional requirements. Simply put, functional requirements are those that would still need to be stated even if a computer with infinite speed, unlimited memory, zero cost, no failures, etc., existed on which to construct the software. All other software product requirements are constraints on automation technologies and are therefore nonfunctional.

Large systems often span more than one subject matter area, or domain. As explained in [9, c6], recursive design shows how nonfunctional requirements in a parent domain can become, or can induce, functional requirements in a child domain. For example, a nonfunctional requirement about user security in a parent banking domain can become or can induce functional requirements in a child security domain. Similarly, cross-cutting nonfunctional requirements about auditing and transaction management in a parent banking domain can become or induce functional requirements in a child auditing domain and a child transaction domain. Decomposing large systems into a set of related domains significantly reduces complexity.

1.9. System Requirements and Software Requirements

The International Council on Systems Engineering (INCOSE) defines a *system* as “an interacting combination of elements to accomplish a defined objective. These include hardware, software, firmware, people, information, techniques, facilities, services, and other support elements” [24].

In some cases, it is either useful or mandatory to distinguish system requirements from software requirements. System requirements apply to larger systems — for example, an autonomous vehicle. Software requirements apply only to an element of

software in that larger system. Some software requirements may be derived from system requirements. (See also [5, c1].) In other cases, the software is itself the system of interest, and hardware and support system are regarded as the platform or infrastructure, so that the system requirements are mostly software requirements.

1.10. Derived Requirements

In practice, *requirements* can be context-sensitive and can depend on perspective. An external stakeholder can impose a scope requirement, and this would be a requirement for the entire project — even if that project involves hundreds of software engineers. An architect's decision to use a pipes-and-filters architecture style would not be a requirement from the perspective of the overall project stakeholders, but a design decision. But that same decision, when seen from the perspective of a sub-team responsible for constructing a particular filter, would be considered a requirement.

The aerospace industry has long used the term *derived requirement* to mean a requirement that was not made by a stakeholder external to the overall project but that was imposed inside the larger development team. The architect's pipes-and-filters decision fits this definition. That choice would be seen as a design decision from the point of view of external stakeholders, but as a requirement for the sub-teams responsible for developing each filter. (See also [9, c4].)

1.11. Software Requirements Activities [1*, c1pp15-18] [2*, s4.2]

Figure 1.3 shows the requirements development and management activities.

Requirements development, as a whole, can be thought of as “reaching an agreement on what software is to be constructed.” In contrast, requirements management can be considered “maintaining that agreement over time.” Each activity is presented in this KA. Requirements development activities are presented as separate

topics, with requirements management presented as a single topic. (See also [5, c1] [6, 2].)

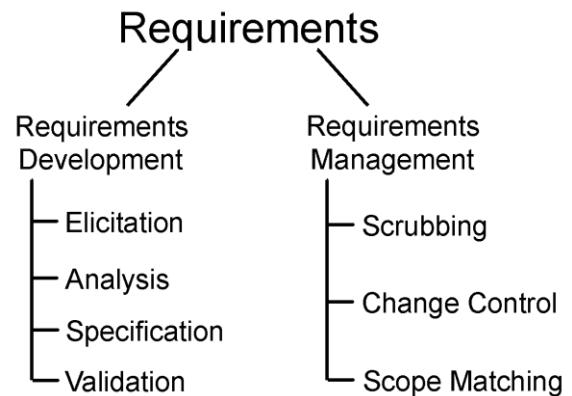


Figure 1.3. Software Requirements Activities

2. Requirements Elicitation [1*, c6-7] [2*, s4.3]

The goal of requirements elicitation is to surface candidate requirements. It is also called *requirements capture*, *requirements discovery* or *requirements acquisition*. As stated earlier, one problem in requirements work on real-world software projects is incompleteness. This can be the result of inadequate elicitation. Although there is no guarantee that a set of requirements is complete, well-executed elicitation helps minimize incompleteness. (See also [5, c2-3] [6, c3-7].)

2.1. Requirements Sources [1*, c6] [2*, s4.3]

Requirements come — can be elicited — from many different sources. All potential requirements sources should be identified and evaluated. A *stakeholder* can be defined as any person, group or organization that:

- is actively involved in the project;
- is affected by the project’s outcome;
- can influence the project’s outcome.

Typical stakeholders for software projects include but are not limited to the following:

- clients — those who pay for the software to be constructed (e.g., organizational management);

- customers — those who decide whether a software product will be put into service;
- users — those who interact directly or indirectly with the software; users can often be further broken down into distinct user classes that vary in frequency of use, tasks performed, skill and knowledge level, privilege level, and so on;
- subject matter experts (SMEs);
- operations staff;
- first-level product support staff;
- relevant professional bodies;
- regulatory agencies;
- special interest groups;
- people who can be negatively affected if the project is successful;
- developers.

Stakeholder classes are groups of stakeholders that have similar perspectives and needs. Working on a software project in terms of stakeholder classes rather than with individual stakeholders can produce important, additional insight.

Many projects benefit from performing a stakeholder analysis to identify as many important stakeholder classes as possible. This reduces the possibility that the requirements are biased toward better-represented stakeholders and away from less well-represented stakeholders. The stakeholder analysis can also inform negotiation and conflict resolution when requirements from one stakeholder class conflict with requirements from another. (See also [5, c3] [6, c3].)

Requirements are not limited to only coming from people. Other, non-person requirements sources can include:

- documentation such as requirements for previous versions, mission statements, concept of operations;
- other systems;
- larger business context including organizational policies and processes;
- computing environment.

2.2. Common Requirements Elicitation Techniques

[1*, c7] [2*, s4.3]

A wide variety of techniques can be used to elicit requirements from stakeholders. Some techniques work better with certain stakeholder classes than others. Common stakeholder elicitation techniques include the following:

- interviews;
- meetings, possibly including brainstorming;
- Joint Application Development (JAD) [13], Joint Requirements Planning (JRP) [14] and other facilitated workshops;
- protocol analysis;
- focus groups
- questionnaires and market surveys
- exploratory prototyping, including low-fidelity and high-fidelity user interface prototyping [1*, c15];
- user story mapping.

Elicitation can be difficult, and the software engineer needs to know that (for example) users might have difficulty describing their tasks, leave important information unstated or be unwilling or unable to cooperate. Elicitation is not a passive activity. Even if cooperative and articulate stakeholders are available, the software engineer must work hard to elicit the right information. Many product requirements are tacit or can be found only in information that has yet to be collected.

Requirements can also be elicited from sources other than stakeholders. Such sources and techniques include the following:

- previous versions of the system;
- defect tracking database for previous versions of the system;
- systems that interface with the system under development;
- competitive benchmarking;
- literature search;
- Quality Function Deployment (QFD)'s House of Quality [15];

- observation, where the software engineer studies the work and the environment where the work is being done;
 - apprenticing, where the software engineer learns by doing the work;
 - usage scenario descriptions;
 - decomposition (e.g., capabilities into epics into features into stories);
 - task analysis [16];
 - design thinking (empathize, define, ideate, prototype, test) [17];
 - ISO/IEC 25010: “System and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models” [27];
 - security requirements, as discussed in the Security KA;
 - applicable standards and regulations.
- be binding, meaning that clients are willing to pay for it and unwilling not to have it;
 - represent true, actual stakeholder needs;
 - use stakeholder vocabulary;
 - be acceptable to all stakeholders.

The overall collection of requirements should be:

- complete, — The requirements adequately address boundary conditions, exception conditions and security needs;
- internally consistent — No requirement conflicts with any other;
- externally consistent — No requirement conflicts with any source material;
- feasible — A viable, cost-effective solution can be created within cost, schedule, staffing, and other constraints.

(See also [5, c3] [6, c4-7].)

3. Requirements Analysis [1*, c8-9]

Requirements are unlikely to be elicited in their final form. Further investigation is usually needed to reveal the full, true requirements suggested by the originally elicited information. Requirements analysis helps software developers understand the meaning and implications of candidate requirements, both individually and in the context of the overall set of requirements.

3.1. Basic Requirements Analysis [1*, c8-9]

The following list of desirable properties of requirements can guide basic requirements analysis. The software engineer seeks to establish any of these properties that do not hold yet. Each requirement should:

- be unambiguous (interpretable in only one way);
- be testable (quantified), meaning that compliance or noncompliance can be clearly demonstrated;

In some cases, an elicited statement represents a solution to be implemented rather than the true problem to be solved. This risks implementing a suboptimal solution. The *5-whys* technique (e.g., [3*, c4]) involves repeatedly asking, “Why is this the requirement?” to converge on the true problem. Repetition stops when the answer is, “If that isn’t done, then the stakeholder’s problem has not been solved.” Often, the true problem is reached in two or three cycles, but the technique is called *5-whys* to incentivize engineers to push it as far as possible.

3.2. Economics of Quality of Service Constraints

[3*]

Quality of service constraints can be particularly challenging. This is generally because engineers do not consider them from an economic perspective [9, c4]. Figure 1.4 illustrates the economic perspective of a typical quality of service constraint, such as capacity, throughput and reliability, where value increases with performance level. This curve is mirrored vertically for quality of service constraints whose value decreases as performance level increases (response time and mean time to repair would be examples).

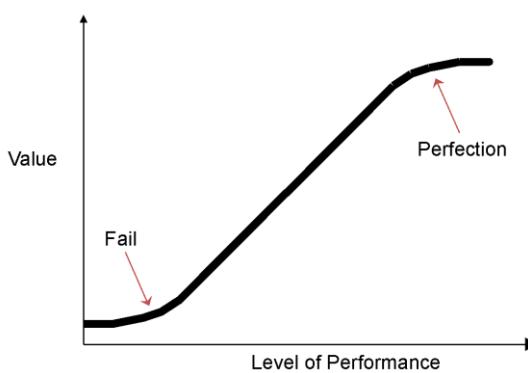


Figure 1.4. Value as a Function of Level of Performance

Over the relevant range of performance levels, the stakeholders have a corresponding value if the system performs at that level. The value curve has two important points:

1. Perfection point — This is the most favorable level of performance, beyond which there is no additional benefit. Even if the system can perform better than the perfection point, the customer cannot use that capacity. For example, a social media system that supports more members than the world population would have this excess capacity.
2. Fail point — This is the least favorable level of performance, beyond which there is no further reduction in benefit. For example, the social media system might need to support at least a minimum market share to be viable as a platform.

A quantified requirement point, even if stated explicitly, is usually arbitrary. It is often based on what a client feels justified requesting, given what they are paying for the software. Even if the software engineers cannot construct a system that fully achieves the stated requirement point, the software typically still has value; it just has less value than the client expected. Further, the ability to exceed the

requirement point can significantly increase value in some cases.

The cost to achieve a given performance level is usually a step function. First, for a given investment level, there is some maximum achievable performance level. Then, additional investment is needed, and that further investment enables performance up to a new, more favorable maximum. Figure 1.5 illustrates the most cost-effective performance level — the performance level with the maximum positive difference between the value at that performance level and the cost to achieve it.

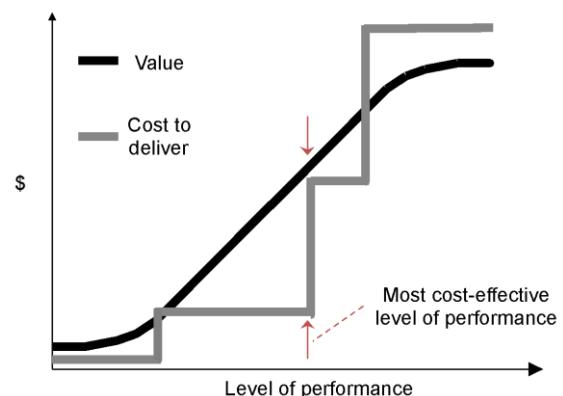


Figure 1.5. Most Cost-Effective Level of Performance

(See the Software Engineering Economics KA or [3*] for more information on performing economic analyses such as this.)

The software engineer should pay particular attention to positive and negative relationships between quality of service constraints (e.g., Figure 14-2 in [1*, c14]). Some quality of service constraints are mutually supporting; improving one's performance level will automatically improve the other's performance level. For example, the more modifiable code is, the more reliable it tends to be, as both modifiability and reliability are, to a degree, a consequence of how clean the code is. On the other hand, the higher the code's speed, the less modifiable it might be, because high speed is often

achieved through optimizations that make the code more complex.

3.3. Formal Analysis [2*, s12.3.2-12.3.3]

Formal analysis has shown benefits in some application domains, particularly high-integrity systems (e.g., [5, c6]). The formal expression of requirements depends on the use of a specification language with formally defined semantics. Formality has two benefits. First, formal requirements are precise and concise, which (in principle) will reduce the possibility for misinterpretation. Second, formal requirements can be reasoned over, permitting desired properties of the specified software to be proved. This permits static validation that the software specified by the requirements does have the properties (e.g., absence of deadlock) that the customer, users and software engineer expect it to have.

This topic is related to Formal Methods in the Software Engineering Models and Methods KA.

3.4. Addressing Conflict in Requirements

When a project has more — and more diverse — stakeholders, conflicts among the requirements are more likely. One particularly important aspect of requirements analysis is identifying and managing such conflicts (e.g., [6, c17]). Once conflicting requirements have been identified, the engineer may consider two different approaches to managing that conflict (and possibly other approaches as well) and determine the most appropriate course of action.

One approach is to negotiate a resolution among the conflicting stakeholders. In most cases, it is unwise for the software engineer to make a unilateral decision, so it becomes necessary to consult with the stakeholders to reach a consensus resolution. It is often also important, for contractual reasons, that such decisions be traceable back to the customer. A specific example is *project scope management* — namely, balancing what's desired in the stated software product requirements with what can be accomplished given the project requirements of cost,

schedule, staffing and other project-level constraints. There are many useful sources for information on negotiation and conflict resolution [25].

Another approach is to apply *product family development* (e.g., [20]). This involves separating requirements into two categories. The first category contains the *invariant requirements*. These are requirements that all stakeholders agree on. The second category contains the *variant requirements*, where conflict exists. The software engineer can focus on understanding the range of variations needed to satisfy all stakeholders. The software can be designed using *design to invariants* to accommodate the invariant requirements and *design for change* to incorporate customization points to configure an instance of the system to best fit relevant stakeholders. In a simple example, some users of a weather application require temperatures displayed in degrees Celsius while others require degrees Fahrenheit.

4. Requirements Specification [1*, c10-14, c20-26] [2*, s4.4, c5]

Requirements specification concerns recording the requirements so they can be both remembered and communicated. Requirements specification might be the most contentious topic in this KA. Debate centers on questions such as:

- should requirements be written down at all?
- if requirements are written down, what form should they take?
- if requirements are written down, should they also be maintained over time?

There are no standard answers to these questions; the answer to each can depend on factors such as the following:

- the software engineer's familiarity with the business domain;
- precedent for this kind of software;

- degree of risk (e.g., probability, severity) of incorrect requirements;
- staff turnover anticipated during the software's service life of the software;
- geographic distribution of the development team members;
- stakeholder involvement over the course of the project;
- whether the use of a packaged solution or open source library is anticipated;
- whether any design or construction will be outsourced;
- the degree of requirements-based testing expected;
- effort needed to use a candidate specification technique;
- accuracy needed from the requirements-based estimates;
- extent of requirements tracing necessary, if any;
- contractual impositions of requirements specification content and format.

As stated in this KA's introduction, the *whats* and *hows* of software requirements work on a project should be determined by the nature of the software constructed, not by the life cycle under which it is constructed. Downstream maintainers should not be able to discern the life cycle used in earlier development from the form of those requirements alone. The chosen life cycle's effect should be limited to the completeness of the requirements at any point in the project. Under a waterfall life cycle, the requirements are expected to be completely specified at the end of the Requirements phase. Under an Agile life cycle, the requirements are expected to change, grow, or be eliminated continuously and not be complete until the project's end.

Some organizations have a culture of documenting requirements; some do not. Dynamic startup projects are often driven by a strong product vision and limited resources; their teams might view requirements documentation as unnecessary overhead. But as these products evolve and mature, software engineers often recognize that they need to

recover the requirements that motivated product features in order to assess the impact of proposed changes. Hence, requirements documentation and change management become important to long-term success. A project's approach to requirements in general, and to requirements specification in particular, may evolve over the service life of that software.

The most basic recommendation for requirements documentation is to base decisions on an *audience analysis*. Who are the different consumers who will need information from a requirements specification? What information will they need? How can that information be packaged and presented so that each consumer can get the information they need with the least effort?

There is a degree of overlap and dependency between requirements analysis and specification. Use of certain requirements specification techniques — particularly model-based requirements specifications — permit and encourage requirements analysis that can go beyond what has already been presented.

Documented software requirements should be subject to the same configuration management practices as the other deliverables of the software life cycle processes. (See the Configuration Management KA for a detailed discussion.) In addition, when practical, the individual requirements are also subject to configuration management, which is generally supported by a requirements management tool. (See Topic 8, Software Requirements Tools.)

There are several general categories of requirements specification techniques, each of which is discussed below. The requirements specification for a given project may also use various techniques. ISO/IEC/IEEE 29148 [26], as well as [1*, c10-14], [5, c4], [6, c16], and many others offer templates for requirements documentation.

4.1. Unstructured Natural Language Requirements Specification [1, c11] [2*, s4.4.1]*

Natural language requirements specifications express requirements in common, ordinary language. Natural language requirements specifications can be unstructured or structured.

A typical unstructured natural language requirements specification is a collection of statements in natural language, such as, “The system shall” For example, business rules are statements that define or constrain some aspect of the structure or the behavior of the business to be automated. “A student cannot register in next semester’s courses if there remain any unpaid tuition fees” is an example of a business rule that serves as a requirement for a university’s course-registration software. Some projects can publish a user manual as a satisfactory requirements specification, although there are limits to how effective this can be. (See also [5, c4] [26].)

4.2. Structured Natural Language Requirements Specification [1*, c8] [2*, s4.4.2]

Structured natural language requirements specifications impose constraints on how the requirements are expressed; the goal is to increase precision and conciseness.

The simplest example might be the actor-action format. The actor is the entity responsible for carrying out the action, and action is what needs to happen. A triggering event might precede the actor, and the action might be followed by an optional condition or qualification. The statement “When an order is shipped, the system shall create an Invoice unless the Order Terms are ‘Prepaid’” uses actor-action format. The triggering event is “When an order is shipped.” The actor is “the system.” The action is “create an Invoice.” The condition/qualification is “except the Order Terms are ‘Prepaid’.”

Another example is a use case specification template, as shown in Figure 1.6. (See [11] for guidelines on writing good use case specifications.)

Use case #66	Use case name: Reserve flight(s)
Triggering event(s)	Customer requests reservation(s) on flight(s)
Parameters	Passenger, itinerary, fare class, payment method(s)
Requires	Legal itinerary, fare class restrictions met
Guarantees	Seat(s) reserved for passenger on itinerary flight(s)
Normal course	Non-FF passenger, all domestic itinerary, Economy fare class, credit/debit card
Alternative course(s)	Is FF passenger: [None, Silver, Gold, Platinum, Elite] Itinerary: [all international, mixed domestic + international] Fare class: [Basic economy, Premium Economy, Business, First] Payment method: [Voucher, FF miles]
Exceptions	C/D card declined, voucher doesn't exist, voucher expired, FF account doesn't exist, insufficient miles in FF account

Figure 1.6. Example of Structured Natural Language Specification for a Single Use Case

The user story format, “As a <user> I want <capability> so that <benefit>” as well as decision tables are other examples. (See also [5, c4] [6, c12, c16] [7, c2-5].)

4.3. Acceptance Criteria-Based Requirements Specification

This general approach includes two specific variants: Acceptance Test Driven Development (ATDD) and Behavior Driven Development (BDD).

ATDD [2*, s3.2.3, s8.2] is a part of the larger Test Driven Development (TDD) approach. (See the Software Testing KA.). The main idea of TDD is that test cases precede construction. Therefore, no new production code is written and no existing code is modified unless at least one test case fails, either at the unit test level or at the acceptance test level. The ATDD process has three steps:

1. A unit of functionality (e.g., a user story) is selected for implementation.
2. One or more software engineers, one or more business domain experts, and possibly one or more QA/test professionals meet — before any production design or construction work is done — to agree on a set of test cases that must pass to show that the unit of functionality has been correctly implemented.
3. At least one of those acceptance test cases must fail on the existing software. The existence of at least one failing test case gives the software engineer(s) permission

to create or modify production code to pass all of the agreed-upon test cases. This step might require several iterations. The code may also be refactored during this step.

When all acceptance test cases have passed, and presumably all unit and integration test cases as well, then the unit of functionality is deemed to have been completely and correctly implemented. The ATDD process returns to step 1, where a new unit of functionality is selected, and the cycle repeats.

ATDD might seem to be a testing technique rather than a requirements specification technique. On the other hand, a test case has the general form of “When given input that looks like X, we expect the software to produce results that look like Y.” The key is the underlined phrase, “we expect the software to produce.” If we simply modify that phrase to say, “the software shall produce,” as in “When given input that looks like X, the software shall produce results that look like Y,” what first looked like a test case now looks like a requirement. Technically, one acceptance test case can encompass more than one single requirement, but the general idea holds that the ATDD test cases are essentially precise, unambiguous statements of requirements.

The BDD approach [19] is slightly more structured, and business domain experts typically prefer it over ATDD because it is less technical in appearance. In BDD, the unit of functionality is described as a user story, in a form such as this: “As a <role>, I want <goal/desire> so that <benefit>.” This leads to the identification and specification of a set of “scenarios” in this form: “Given <some context> [and <possibly more context>], when <stimulus> then <outcome> [and <possibly more outcomes>].”

If the story is “As a bank customer, I want to withdraw cash from the Automated Teller Machine (ATM) so that I can get money without going to the bank,” one scenario could be that “the account has a sufficient balance.” This scenario could be detailed as “Given the account balance is \$500, and the customer’s bank card is valid, and the automated

teller machine contains enough money in its cash box, when the Account Holder requests \$100, then the ATM should dispense \$100 and the account balance should be \$400, and the customer’s bank card should be returned.”

Another scenario could be that “the account has an insufficient balance” and could be detailed as “Given the account balance is \$50, and the customer’s bank card is valid, and the automated teller machine contains enough money in its cash box, when the Account Holder requests \$100, then the ATM should not dispense any money, and the ATM should say there is an insufficient balance, the balance should remain at \$50, and the customer’s bank card should be returned.”

The goal of BDD is to have a comprehensive set of scenarios for each unit of functionality. In the withdrawing cash situation, additional scenarios for “The Bank Customer’s bank card has been disabled” and “The ATM does not contain enough money in its cash box” would be necessary.

The acceptance test cases are obvious from the BDD scenarios.

Acceptance criteria-based requirements specification directly addresses the requirements ambiguity problem. Natural languages are inherently ambiguous, but test case language is not. In acceptance-based criteria requirements specification, the requirements are written using test case language, which is very precise. On the other hand, this does not inherently solve the incompleteness problem. However, combining ATDD or BDD with appropriate functional test coverage criteria, such as Domain Testing, Boundary Value Analysis and Pairwise Testing (see the Software Testing KA), can reduce the likelihood of requirements incompleteness. (See also [9, c1, c12].)

4.4. Model-Based Requirements Specification [1, c12] [2*, c5] [4*]*

Another approach to avoiding the inherent ambiguity of natural languages is to use modeling languages such as selected elements of the Unified

Modeling Language™ (UML) or Systems Modeling Language™ (SysML). Much like the blueprints used in building construction, these modeling languages can be used in a computing technology-free manner to precisely and concisely specify functional requirements [9, c1-2]. This topic is closely related to the Software Engineering Models and Methods KA. Requirements models fall into two general categories:

1. Structural models for specifying policies to be enforced: These are logical class models as described in, for example, [9, c8]. They are also called conceptual data models, logical data models and entity-relationship diagrams.
2. Behavioral models for specifying processes to be carried out: These models include use case modeling as described in [9, c7], interaction diagrams as described in [9, c9] and state modeling as described in [9, c10]. Other examples are UML activity diagrams and data-flow modeling, as described in [1*, c12-13], [8], [10] and [18].

Model-based requirements specifications vary in the degree of model formality. Consider the following:

- Agile modeling (see, for example, [10]) is the least formal. Agile models can be little more than rough sketches whose goal is to communicate important information rather than demonstrate proper use of modeling notations. In this type of modeling, the effect of the communication is considered more important than the form of the communication.
- Semiformal modeling, for example [9, c6-12], provides a definition of the modeling language semantics ([9, Appendix L]), but that definition has not been formally proved to be complete and consistent.
- Formal modeling, for example, Z, the Vienna Development Method (VDM), Specification and Description Language (SDL) and [5, c6] have very precisely defined semantics that allow specifications to be mechanically analyzed for the presence or absence of specific properties to

help avoid critical reasoning errors. The term *correctness by construction* has been used for development in this context. (See the Formal Methods section in the Software Engineering Models and Methods KA.)

Generally, the more formal a requirements model is, the less ambiguous it is, so software engineers are less likely to misinterpret the requirements. More formal requirements models can also be:

- more concise and compact;
- easier to translate into code, possibly mechanically;
- used as a basis for deriving acceptance test cases.

One important message in [4*] is that while formal modeling languages are stronger than semiformal and Agile modeling, formal notations can burden both the model creator and human readers. Wing's compromise is to use formally defined underpinnings (e.g., in Z) for surface syntaxes that are easier to read and write (e.g., UML statecharts).

4.5. Additional Attributes of Requirements [1, c27pp462-463]*

Over and above the basic requirements statements already described, documenting additional attributes for some or all requirements can be useful. This supplemental detail can help software engineers better interpret and manage the requirements [6, c16]. Possible additional attributes include the following:

- tag to support requirements tracing;
- description (additional details about the requirement);
- rationale (why the requirement is important);
- source (role or name of the stakeholder who imposed this requirement);
- use case or relevant triggering event;
- type (classification or category of the requirement — e.g., functional, quality of service);
- dependencies;

- conflicts;
- acceptance criteria;
- priority (see Requirements Prioritization later in this KA);
- stability (see Requirements Stability and Volatility later in this KA);
- whether the requirement is common or a variant for product family development (e.g., [20]);
- supporting materials;
- the requirement's change history.

Gilb's Planguage (short for Planning Language) [7] recommends attributes such as scale, meter, minimum, target, outstanding, past, trend and record.

4.6. Incremental and Comprehensive Requirements Specification

Projects that explicitly document requirements take one of two approaches. One can be called *incremental specification*. In this approach, a version of the requirements specification contains only the differences — additions, modifications and deletions — from the previous version. An advantage of this approach is that it can produce a smaller volume of written specifications.

The other approach can be called *comprehensive specification*. In this approach, each version's requirements specification contains all requirements, not just changes from the previous version. An advantage of this approach is that a reader can understand all requirements in a single document instead of having to keep track of cumulative additions, modifications and deletions across a series of specifications.

Some organizations combine these two approaches, producing intermediate releases (e.g., x.1, x.2 and x.3) that are specified incrementally and major releases (e.g., 1.0, 2.0 and 3.0) that are specified comprehensively. The reader never needs to go any further back than the requirements specifications for the last major release to obtain the complete set of specifications.

5. Requirements Validation [1*, c17] [2*, s4.5]

Requirements validation concerns gaining confidence that the requirements represent the stakeholders' true needs as they are currently understood (and possibly documented). Key questions include the following:

- do these represent all requirements relevant at this time?
- are any stated requirements not representative of stakeholder needs?
- are these requirements appropriately stated?
- are the requirements understandable, consistent and complete?
- does the requirements documentation conform to relevant standards?

Three methods for requirements validation tend to be used: requirements reviews, simulation and execution, and prototyping. (See also [5, c5] [6, c17] [9, c12].)

5.1. Requirements Reviews [1, c17pp332-342] [2*, c4p130]*

The most common way to validate is by reviewing or inspecting a requirements document. One or more reviewers are asked to look for errors, omissions, invalid assumptions, lack of clarity and deviation from accepted practice. Review from multiple perspectives is preferred:

- clients, customers and users check that their wants and needs are completely and accurately represented;
- other software engineers with expertise in requirements specification check that the document is clear and conforms to applicable standards;
- software engineers who will do architecture, design or construction of the software that satisfies these requirements check that the document is sufficient to support their work.

Providing checklists, quality criteria or a “definition of done” to the reviewers can guide them to focus on specific aspects of the requirements specification. (See Reviews and Audits in the Software Quality KA.)

5.2. Simulation and Execution

Nontechnical stakeholders might not want to spend time reviewing a specification in detail. Some specifications can be subjected to simulation or actual execution in place of or in addition to human review. To the extent that the requirements are formally specified (e.g., in a model-based specification), software engineers can hand interpret that specification and “execute” the specification. Given a sufficient set of demonstration scenarios, stakeholders can be convinced that the specification defines their policies and processes completely and accurately. (See [9, c12].)

5.3. Prototyping [1, c17p342] [2*, c4p130]*

If the requirements specification is not in a form that allows direct simulation or execution, an alternative is to have a software engineer build a prototype that concretely demonstrates some important dimension of an implementation. This demonstrates the software engineer’s interpretation of those requirements.

Prototypes can help expose software engineers’ assumptions and, where needed, give useful feedback on why they are wrong. For example, a user interface’s dynamic behavior might be better understood through an animated prototype than through textual description or graphical models. However, a danger of prototyping is that cosmetic issues or quality problems with the prototype can distract the reviewers’ attention from the core underlying functionality. Prototypes can also be costly to develop. However, if a prototype helps engineers avoid the waste caused by trying to satisfy erroneous requirements, its cost can be more easily justified.

6. Requirements Management Activities [1*, c27-28] [2*, s4.6]

Requirements development, as a whole, can be thought of as “reaching an agreement on what software is to be constructed.” (See Figure 1.3.) In contrast, requirements management can be thought of as “maintaining that agreement over time.” This topic examines requirements management. (See also [5, c9].)

6.1. Requirements Scrubbing

The goal of requirements scrubbing [22, c14, c32] is to find the smallest set of simply stated requirements that will meet stakeholder needs. Doing so will reduce the size and complexity of the solution, thus minimizing the effort, cost and schedule to deliver it. Requirements scrubbing involves eliminating requirements that:

- are out of scope;
- would not yield an adequate return on investment;
- are not that important.

Another important part of the process is to simplify unnecessarily complicated requirements.

In waterfall and other plan-based life cycles, requirements scrubbing can be coordinated with requirements reviews for validation; scrubbing should occur just before the validation review. In Agile life cycles, scrubbing happens implicitly in iteration planning; only the highest-priority requirements are brought into a sprint (iteration).

6.2. Requirements Change Control [1, c28] [2*, s4.6]*

Change control is central to managing requirements. This topic is closely linked to the Software Configuration Management KA. (Refer to that chapter for more information.)

Projects using waterfall or other plan-based life cycles should have an explicit requirements change control process that includes:

- a means to request changes to previously agreed-upon requirements;
- an optional impact analysis stage to more thoroughly examine benefits and costs of a requested change;
- a responsible person or group who decides to accept, reject, or defer each requested change;
- a means to notify all affected stakeholders of that decision;
- a means to track accepted changes to closure.

All stakeholders must understand and agree that accepting a change means accepting its impact on schedule, resources and/or commensurate change in scope elsewhere in the project.

In contrast, requirements change management happens implicitly in Agile life cycles. In these life cycles, any request to change previously agreed-upon requirements becomes just another item on the product backlog. A request will only become “accepted” when it is prioritized highly enough to make it into an iteration (a sprint). (See also [5, c9] [22, c17].)

6.3. Scope Matching

Scope matching [22, c14] involves ensuring that the scope of requirements to architect, design and construct does not exceed any cost, schedule or staffing constraints on the project. When requirements scope exceeds the cost, schedule or staffing constraints, then either that scope must be reduced (presumably by removing a sufficient number of the lowest-priority requirements), capacity must be increased (by extending the schedule or increasing the budget and/or staffing), or some appropriate combination thereof must be negotiated.

In waterfall and other plan-based life cycles, scope matching can be coordinated with requirements validation; the scope matching should occur just before the validation review. In Agile life cycles, as long as some variant of *velocity-based sprint*

planning is done, then the only work allowed into a sprint/iteration will be the work that can reasonably be expected to be completed during that sprint/iteration.

7. Practical Considerations

7.1. Iterative Nature of the Requirements Process [2*, s4.2]

Requirements for typical software not only have wide breadth; they must also have significant depth. The tension created by simultaneous breadth-wise and depth-wise requirements in real-world projects often prompts teams to perform requirements activities iteratively. At some points, elicitation and analysis favor expanding the breadth of requirements knowledge, while at other points, expanding the depth is called for. In practice, it is highly unlikely that all requirements work can be done in a single pass through the subject matter. (See also [6, c2, c9].)

7.2. Requirements Prioritization [1*, c16]

Prioritizing requirements is useful throughout a software project because it helps focus software engineers on delivering the most valuable functionality soonest. It also helps support intelligent trade-off decisions involving conflict resolution and scope matching. Prioritized requirements also help in maintenance beyond the initial development project itself. Defects raised against higher-priority requirements should probably be repaired before defects raised against lower-priority ones.

A variety of prioritization schemes are available. Answering a few key questions can help engineers choose the best approach. The first question is “What factors are relevant in determining the priority of one requirement over another?” The following factors might be relevant to a project:

- value; desirability; client, customer and user satisfaction;
- undesirability; client, customer and user dissatisfaction (Kano model, below);
- cost to deliver;

- cost to maintain over the software's service life;
- technical risk of implementation;
- risk that users will not use it even if implemented.

The Kano model, which underlies [6, c17], shows that considering only value, desirability or satisfaction can lead to erroneous priorities. A better understanding of priorities comes from considering how unhappy stakeholders would be if that requirement were not satisfied. For example, consider a project to develop an email client. Two candidate requirements might relate to:

1. Having an effective spam filter
2. Handling attachments on emails

Prioritization must weigh both the satisfaction users will experience from having certain features and the dissatisfaction they will experience if they lack certain features. For example, users are more likely to be happy with an effective spam filter than with the ability to handle attachments, so the spam filter would be given a higher priority based on the satisfaction criterion. On the other hand, the inability to handle attachments would make many users extremely unhappy — much more so than not having an effective spam filter. When considering happiness, or satisfaction, from implementing features combined with unhappiness (or dissatisfaction) from not implementing certain features, developers would generally give handling attachments a higher priority than the effective spam filter.

The second key question is “How can we convert the set of relevant factors into an expression of priority?” The formula

$$\text{Priority} = \frac{(\text{Value} * (1 - \text{Risk}))}{\text{Cost}}$$

is just one example of an *objective function* to do so. The choice of measurement schemes for the relevant factors can impose constraints on the objective

function. (See Measurement Theory in Computing Foundations).

Once the priority of the requirements has been determined, those priorities must be specified in a way that can be communicated to all stakeholders. Several ways to do this are possible, including the following:

- enumerated scale (e.g., must have, should have, nice to have);
- numerical scale (e.g., 1 . . . 10);
- Lists that sort the requirements in decreasing priority order.

Effective requirement prioritization focuses on finding groups of requirements with similar priorities rather than creating overly rigorous measurement scales or debating small differences.

7.3. Requirements Tracing [1*, c29]

Requirements tracing can serve two potentially useful purposes. One is to serve as an accounting exercise that documents consistency between pairs of related project work products. An important question might be “For each identified software requirement, are there identified design elements intended to satisfy it?” If no identified design elements can be found, then either that requirement is not satisfied in that design or the design is correct and one or more stated requirements can be deleted. Similarly, “For each identified design element, are there identified requirements that cause it to exist?” If no identified requirements can be found, then either that design element is unnecessary or the stated requirements are incomplete.

The other purpose is to assist in impact analysis of a proposed requirement change. If a particular system requirement were to change, for example, that system requirement could be traced to its linked software requirements. Not all linked software requirements would need to change. But each software requirement that would be affected could be traced to its linked design elements. Again, not all linked design elements would need to change.

But each design element affected could be traced to the linked code. The affected software requirements, design elements and code units could also be traced to their linked test cases for further impact analysis. This helps establish a “footprint” for the volume of work needed to incorporate that change to the system requirement.

Software requirements can be traced back to source documentation such as system requirements, standards documents and other relevant specifications. Software requirements can also be traced forward to design elements and requirements-based test cases. Finally, software requirements can also be traced forward to sections in a user manual describing the implemented functionality. (See also [23].)

7.4. Requirements Stability and Volatility [2*, s4.6]
 Some requirements are very stable; they will probably never change over the software’s service life. Some requirements are less stable; they might change over the service life but might not change during the development project. For example, in a banking application, requirements for functions to calculate and credit interest to customers’ accounts are likely to be more stable than requirements to support different tax-free accounts. The former reflects a banking domain’s fundamental feature (that accounts can earn interest). At the same time, the latter may be rendered obsolete by a change in government legislation. Finally, some requirements can be very unstable; they can change during the project — possibly more than once. It is useful to assess the likelihood that a requirement will change in a given time. Identifying potentially volatile requirements helps the software engineer establish a design more tolerant of change, (e.g., [20]). (See also [9, c4].)

7.5. Measuring Requirements

[1*, c19]

As a practical matter, it may be useful to have some concept of the *volume* of the requirements for a particular software product. This number is useful in evaluating the *size* of a new development project or the size of a change in requirements and in

estimating the cost of development or maintenance tasks (e.g., [9, c23]), or simply for use as the denominator in other measurements. Functional size measurement (FSM) is a technique for evaluating the size of a body of functional requirements.

Additional information on size measurement and standards can be found in the Software Engineering Process KA.

Many quality indicators have been developed that can be used to relate the quality of software requirements specification to other project variables such as cost, acceptance, performance, schedule and reproducibility. Quality indicators for individual software requirements and a requirements specification document as a whole can be derived from the desirable properties discussed in Section 3.1, Basic Requirements Analysis, earlier in this KA.

7.6. Requirements Process Quality and Improvement

[1*, c31]

This topic concerns assessing the quality and improvement of the requirements process. Its purpose is to emphasize the key role of the requirements process in a software product’s cost and timeliness and in customer satisfaction. Furthermore, it helps align the requirements process with quality standards and process improvement models for software and systems. Process quality and improvement are closely related to both the Software Quality KA and Software Engineering Process KA, comprising the following:

- requirements process coverage by process improvement standards and models;
- requirements process measures and benchmarking;
- improvement planning and implementation;
- security/CIA (confidentiality, integrity, and availability) improvement/planning and implementation.

8. Software Requirements Tools

[1*, c30]

Tools that help software engineers deal with software requirements fall broadly into three categories: requirements management tools, requirements modeling tools and functional test case generation tools, as discussed below.

8.1. Requirements Management Tools [1, c30pp506-510]*

Requirements management tools support various activities, including storing requirements attributes, tracing, document generation and change control. Indeed, tracing and change control might only be practical when supported by a tool. Because requirements management is fundamental to good requirements practice, many organizations have invested in tools. However, many more manage their requirements in more ad hoc and generally less satisfactory ways (e.g., spreadsheets). (See also [5, c8].)

8.2. Requirements Modeling Tools [1, c30p506] [2*, s12.3.3]*

At a minimum, a requirements modeling tool supports creating, modifying and publishing model-based requirements specifications. Some tools extend that by also providing static analysis (e.g., syntax correctness, completeness and consistency). Formal analysis requires tool support to be practicable for anything other than trivial systems, and tools generally fall into two categories: theorem provers or model checkers. In neither case can proof be fully automated, and the competence in formal reasoning needed to use the tools restricts the wider formal analysis. Some tools also dynamically execute a specification (simulation).

8.3. Functional Test Case Generation Tools

The more formally defined a requirements specification language is, the more likely it is that functional test cases can be at least partially derived mechanically. For example, converting BDD scenarios into test cases is not difficult. Another example involves state models. Positive test cases can be derived for each defined transition in that kind of model. Negative test cases can be derived from the state and event combinations that do not appear. (See Section 8.2, Testing Tools in the Testing KA, for more information.) A process for deriving test cases from UML requirements models can be found in [9, c12].

In the most general case, such tools can only generate test case inputs. Determining an expected result is not always possible. Additional business domain expertise might be necessary.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	Wiegers 2013 [1*]	Sommerville 2018 [2*]	Tockey 2005 [3*]	Wing 1990 [4*]

1. Software Requirements Fundamentals				
1.1. Definition of a Software Requirement	c1pp5-6	c4p102		
1.2. Categories of Software Requirements	c1pp7-12	s4.1		
1.3. Software Product Requirements and Software Project Requirements	c1pp14-15			
1.4. Functional Requirements	c1p9	s4.1.1		
1.5. Nonfunctional Requirements	c1pp10-11	s4.1.2		
1.6. Technology Constraints				
1.7. Quality of Service Constraints				
1.8. Why Categorize Requirements This Way?				
1.9. System Requirements and Software Requirements				
1.10. Derived Requirements				
1.11. Software Requirements Activities	c1pp15-18	s4.2		
2. Requirements Elicitation				
2.1. Requirements Sources	c6	s4.3		
2.2. Common Requirements Elicitation Techniques	c7	s4.3		
3. Requirements Analysis				
3.1. Basic Requirements Analysis	c8-9			
3.2. Economics of Quality of Service Constraints			c1-27	
3.3. Formal Analysis		s12.3.2-12.3.3		
3.4. Addressing Conflict in Requirements				
4. Requirements Specification				
4.1. Unstructured Natural Language Requirements Specification	c11	s4.4.1		
4.2. Structured Natural Language Requirements Specification	c8	s4.4.2		
4.3. Acceptance Criteria-Based Requirements Specification		s3.2.3, s8.2		
4.4. Model-Based Requirements Specification	c12	c5		pp8-11
4.5. Additional Attributes of Requirements	c27pp462-463			
4.6. Incremental and Comprehensive Requirements Specification				
5. Requirements Validation				

5.1. Requirements Reviews	c17pp332-342	c4p130		
5.2. Simulation and Execution				
5.3. Prototyping	c17p342	c4p130		
6. Requirements Management Activities				
6.1. Requirements Scrubbing				
6.2. Requirements Change Control	c28	s4.6		
6.3. Scope Matching				
7. Practical Considerations				
7.1. Iterative Nature of the Requirements Process		s4.2		
7.2. Requirements Prioritization	c16			
7.3. Requirements Tracing	c29			
7.4. Requirements Stability and Volatility		s4.6		
7.5. Measuring Requirements	c19			
7.6. Requirements Process Quality and Improvement	c31			
8. Software Requirements Tools				
8.1. Requirements Management Tools	c30pp506-510			
8.2. Requirements Modeling Tools	c30p506	s12.3.3		
8.3. Functional Test Case Generation Tools				

FURTHER READING

P. LaPlante, *Requirements Engineering for Software and Systems* [5].

This book is one potential alternative to [1*], offering a comprehensive discussion of software requirements.

S. Robertson and J. Robertson, *Mastering the Requirements Process: Getting Requirements Right* [6].

This book is another potential alternative to [1*], offering a comprehensive discussion of software requirements.

T. Gilb, *Competitive Engineering: A Handbook for Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage* [7].

This book presents a unique perspective on requirements, emphasizing requirements precision and completeness along with a strong business value-driven motivation.

K. Wiegers, *Software Development Pearls: Lessons from Fifty Years of Software Experience* [21].

This book is a compendium of important but often unrecognized key success factors based on

Dr. Wiegers' extensive real-world experience. Chapter 2 is specific to software requirements.

REFERENCES

R. Fisher and W. Ury, *Getting to Yes* [25].

This book is a classic reference on principled negotiation and conflict resolution that serves as one good basis for addressing inevitable conflict in software requirements when there are multiple stakeholders.

N. Ahmad, *Effects of Electronic Communication on the Elicitation of Tacit Knowledge in Interview Techniques for Small Software Developments* [29].

This doctoral thesis shows how using four different types of electronic communication tools to discuss interview agenda details with interviewees before conducting semi-structured interviews for requirements elicitation improved elicitation of tacit (hidden) knowledge.

- [1*] K. E. Wiegers and J. Beatty, *Software Requirements*, 3rd ed., Redmond, WA: Microsoft Press, 2013.
- [2*] I. Sommerville, *Software Engineering*, 10th ed., New York: Addison-Wesley, 2018.
- [3*] S. Tockey, *Return on Software: Maximizing the Return on Your Software Investment*, Boston, MA: Addison-Wesley, 2005.
- [4*] J. M. Wing, "A Specifier's Introduction to Formal Methods," *Computer*, vol. 23, no. 9, 1990, pp. 8, 10-23.
- [5] P. LaPlante, *Requirements Engineering for Software and Systems*, 3rd ed., Boca Raton, FL: CRC Press, 2018.
- [6] S. Robertson and J. Robertson, *Mastering the Requirements Process: Getting Requirements Right*, Upper Saddle River, NJ: Addison-Wesley, 2013.
- [7] T. Gilb, *Competitive Engineering: A Handbook for Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage*, Oxford, UK: Elsevier Butterworth-Heinemann, 2005.
- [8] E. Yourdon, *Modern Structured Analysis*, Englewood Cliffs, NJ: Prentice-Hall, 1989.
- [9] S. Tockey, *How to Engineer Software*, Hoboken, NJ: Wiley, 2019.
- [10] S. Ambler, *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*, Hoboken, NJ: Wiley, 2002.

- [11] A. Cockburn, *Writing Effective Use Cases*, Upper Saddle River, NJ: Addison-Wesley, 2000.
- [12] L. Constantine and L. Lockwood, *Software for Use*, Reading, MA: Addison-Wesley, 2000.
- [13] J. Wood and D. Silver, *Joint Application Development*, New York, NY: Wiley, 1995.
- [14] E. Gottesdiener, *Requirements by Collaboration*, Boston, MA: Addison-Wesley, 2002.
- [15] J. Terninko, *Step by Step QFD*, 2nd ed., Boca Raton, FL: CRC Press, 1997.
- [16] G. Salvendy, *Handbook of Human Factors*, 4th ed., Hoboken, NJ: Wiley, 2012.
- [17] T. Brown and B. Katz, *Change by Design: How Design Thinking Transforms Organizations and Inspires Innovation*, Revised and updated ed., New York, NY: Harper Collins, 2019.
- [18] S. McMenamin and J. Palmer, *Essential Systems Analysis*, New York, NY: Yourdon Press, 1984.
- [19] J. Smart, *BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle*, Shelter Island, NY: Manning Publications, 2015.
- [20] D. Weiss and C. Lai, *Software Product-Line Engineering: A Family-Based Software Development Process*, Reading, MA: Addison-Wesley, 1999.
- [21] K. Wiegers, *Software Development Pearls: Lessons from Fifty Years of Software Experience*, Boston, MA: Addison-Wesley Professional, 2021.
- [22] S. McConnell, *Rapid Development*, Redmond, WA: Microsoft Press, 1996.
- [23] O. Gotel and C. W. Finkelstein, "An Analysis of the Requirements Traceability Problem," presented at the Proceedings of the 1st International Conference on Requirements Engineering, 1994.
- [24] INCOSE, *Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities*, 3.2.2 ed., San Diego, US: International Council on Systems Engineering, 2012.
- [25] R. Fisher and W. Ury, *Getting to Yes*, 3rd ed., New York, NY: Penguin, 2011.
- [26] ___, ISO/IEC/IEEE 29148 "Systems and software engineering – Life cycle processes – Requirements engineering," *International Standards Organization*, 2018.
- [27] ___, ISO/IEC 25010: "System and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models," *International Standards Organization*, 2011.
- [28] ___, Software and Systems Engineering Vocabulary, https://pascal.computer.org/sev_display/index.action.
- [29] N. Ahmad, *Effects of Electronic Communication on the Elicitation of Tacit Knowledge in Interview Techniques for Small Software Developments*, doctoral thesis, University of Huddersfield, 2021.

CHAPTER 2

Software Architecture

ACRONYMS

AD	architecture description
ADL	architecture description language
API	application programming interface
ASR	architecturally significant requirement
IDL	interface description language
MVC	model view controller

INTRODUCTION

This chapter considers software architecture from several perspectives: concepts; representation and work products; context, process and methods; and analysis and evaluation.

In contrast to the previous edition, this edition creates a Software Architecture Knowledge Area (KA), separate from the Software Design KA, because of the significant interest and growth of the discipline since the 1990s.

BREAKDOWN OF TOPICS FOR SOFTWARE ARCHITECTURE

The breakdown of topics for the Software Architecture KA is shown in Fig. 1.

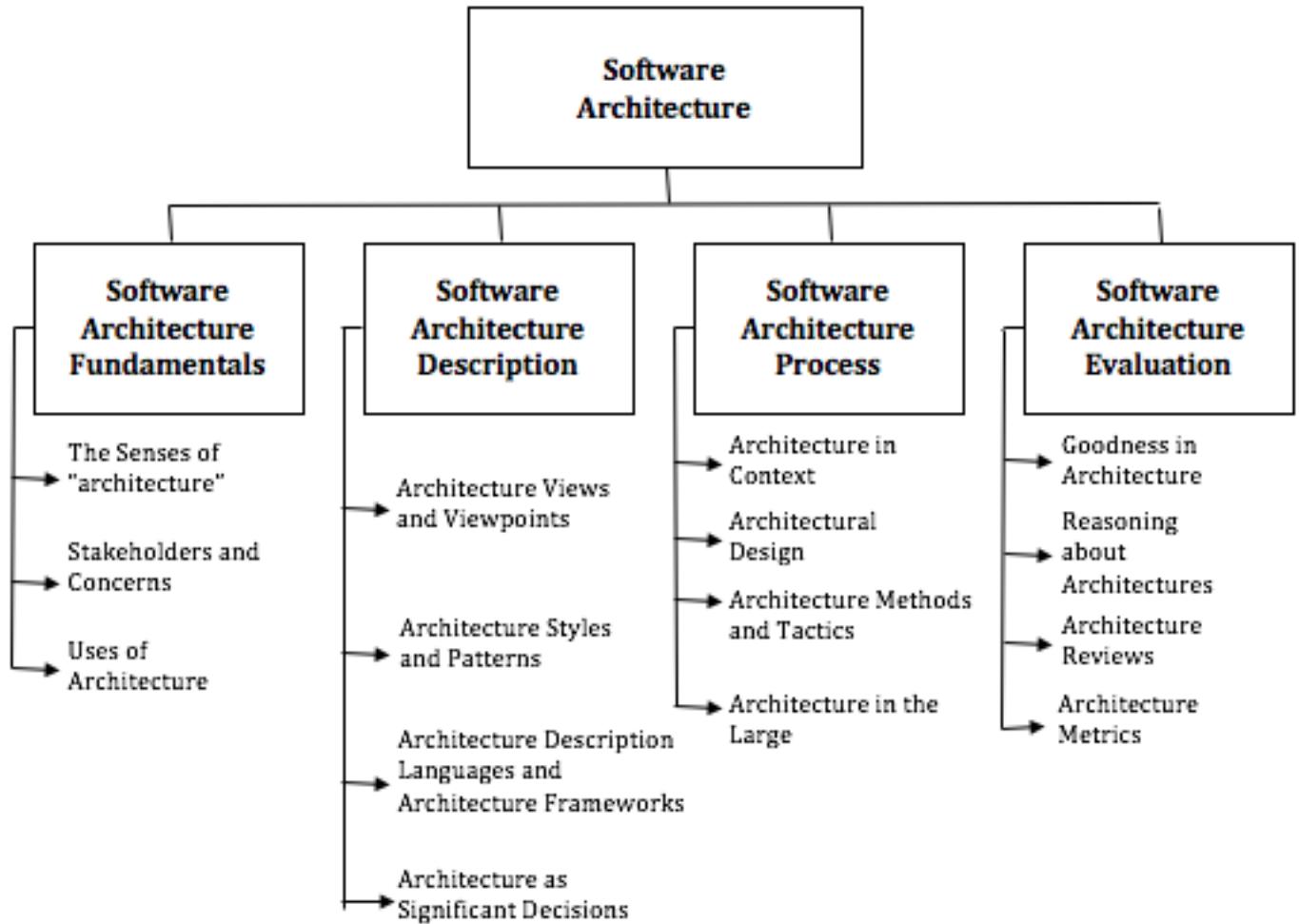


Fig. 1. Breakdown of Topics for the Software Architecture KA

1 SOFTWARE ARCHITECTURE FUNDAMENTALS

1.1 The Senses of “Architecture”

Software engineering and related disciplines use many senses of “architecture”. First, “architecture” often refers to a discipline: the art and science of constructing things — in this case, software-intensive systems. The discipline involves concepts, principles, processes and methods the community has discovered and adopted.

Second, *architecture* refers to the various processes through which that discipline is realized. In this KA, we distinguish *architecture design* as a specific phase in the life cycle encompassing a particular set of activities, and we distinguish it from the wider *architecting* processes that span the life cycle. Both are discussed in [topic Software Architecture Processes](#).

Third, “architecture” refers to the *outcome* of applying architectural design discipline and processes to devise architectures for software systems. Architectures as outcomes are expressed in *architecture descriptions*. This is discussed in [topic Software Architecture Description](#). The concept of architecture has evolved, and many definitions are in use today. One early definition of architecture, from 1990, emphasized software structure:

Architecture. The organizational structure of a system or component. [from: IEEE Std 610.12–1990, *IEEE Glossary of Software Engineering Terminology*]

This definition did not do justice to evolving thinking about architecture; e.g., this definition does not allow us to distinguish the detailed design of a module from its Makefile. Either example reflects an *organizational structure* of the software system or component but should not be considered architecture. Moreover, emphasis on the structure was often limited to the code’s structure and failed to encompass all the structures of the software system:

The software architecture of a system is the set of structures needed to reason about the system. These structures comprise software elements, relations among them, and properties of both. [2*]

During the mid-1990s, however, software architecture emerged as a broader discipline involving a more generic study of software structures and architectures. Many software system structures are not directly reflected in the code structure. Both types of structure have implications for the system as a whole: What behaviors is the system capable of? What interactions does it have with other systems? How are properties like safety and security handled? The recognition that software contains many different structures has prompted discussion of a number of interesting concepts about software architecture (and software design more generally) leading to current definitions such as:

architecture (of a system). fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution [17]

Key ideas in that definition are the following:
(1) Architecture is about what is *fundamental* to a software system; not every element, interconnection, or interface is

considered fundamental. (2) Architecture considers a system *in its environment*. Much like building architecture, software architecture is outward-looking; it considers a system’s context beyond its boundaries to consider the people, organizations, software, hardware and other devices with which the system must interact.

1.2 Stakeholders and Concerns

A software system has many *stakeholders* with varying roles and interests relative to that system. These varying interests are termed *concerns*, following Dijkstra’s *separation of concerns*:

Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one’s subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask ourselves whether, and if so: why, the program is desirable. But nothing is gained — on the contrary! — by tackling these various aspects simultaneously. It is what I sometimes have called “the separation of concerns”, which, even if not perfectly possible, is yet the only available technique for effective ordering of one’s thoughts, that I know of. This is what I mean by “[focusing] one’s attention upon some aspect”: it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect’s point of view, the other is irrelevant. It is being one- and multiple-track-minded simultaneously. [10]

What is fundamental about a system varies according to stakeholders’ concerns and roles. The software structures, therefore, also vary with stakeholder roles and concerns. (See also [topic Design Methods in Software Design KA](#).)

A software system’s customer is most interested in when the system will be ready and how much it will cost to build and operate. Users are most interested in what it does and how to use it. Designers and programmers building the system have their own concerns, such as whether an algorithm will meet the system requirements. Those responsible for ensuring the system is safe to operate have different concerns.

Concerns encompass a broad range of issues, possibly pertaining to any influence on a system in its environment, including developmental, technological, business, operational, organizational, political, economic, legal, regulatory, ecological and social influences. Like software requirements, they may be classified in terms of functional, non-functional or constraint. See [Software Requirements KA](#). Concerns manifest in various familiar forms, including requirements, quality attributes or “ilities”, emergent properties (which may be either desired or prohibited) and various kinds of constraints (as listed above). See [Software Quality KA](#). [Topic 2, Software Architecture Description](#), shows how concerns shape architecture and the work products describing those architectures. Example of concerns are depicted in Fig. 2.

affordability, agility, assurance, autonomy, availability, behavior, business goals and strategies, complexity, compliance

with regulation, concurrency, control, cost, data accessibility, deployability, disposability, energy efficiency, evolvability, extensibility, feasibility, flexibility, functionality, information assurance, inter-process communication, interoperability, known limitations, maintainability, modifiability, modularity, openness, performance, privacy, quality of service, reliability, resource utilization, reusability, safety, scalability, schedule, security, system modes, software structure, subsystem integration, sustainability, system features, testability, usability, usage, user experience

Fig. 2. Examples of Architectural Concerns

1.3 Uses of Architecture

A principal use of a software system’s architecture is to give those working with it a shared understanding of the system to guide its design and construction. An architecture also serves as a preliminary conception of the software system that provides a basis to analyze and evaluate alternatives. A third common usage is to enable reverse engineering (or *reverse architecting*) by helping those working with it to understand an existing software system before undertaking maintenance, enhancement or modification. To support these uses, the architecture should be documented (see [topic Software Architecture Description](#)).

Conway’s Law posits that “organizations which design systems . . . are constrained to produce designs which are copies of the communication structures of these organizations.” [9] This suggests that architectures often mirror the structure of the organizations that developed them. Depending on the software system and the organization, this can be a strength or a weakness. The architecture can enhance communication within a large team or compromise it. Each part of the organization can base its planning, costing and scheduling activities upon its knowledge of the architecture. Creating a well-planned and documented architecture is one approach to increasing the applicability and reusability of software designs and components. The architecture forms the basis for design families of programs or software product lines. This can be done by identifying commonalities among members of such families and by designing reusable and customizable components to account for the variability among family members.

2 SOFTWARE ARCHITECTURE DESCRIPTION

In [topic 1, Software Architecture Fundamentals](#), a software architecture was defined as the fundamental concepts or properties of a software system in its environment. But each stakeholder can have a different notion of what is fundamental to that software system, given their perspective. Having a mental model of a system’s architecture is perhaps fine for small systems and for individuals working alone. However, for large, complex systems developed and operated by teams, a tangible representation is invaluable, especially as the conception of the system evolves, and as people join or leave the team. Having a concrete representation as a work product can also serve as a basis to analyze the architecture, organize its design and guide its implementation. These work products are called *architecture descriptions* (ADs).

An AD documents an architecture for a software system. It is targeted to those stakeholders of the system who have

concerns about the software system which are answered by the architecture. As noted in [topic 1, Software Architecture Fundamentals](#), a primary audience comprises the designers, engineers and programmers whose concerns pertain to constructing the system. For these stakeholders, the AD serves as a *blueprint* to guide the construction of the software system. For others, the AD is a basis for their work—for example, testing and quality assurance, certification, deployment, operation, and maintenance and future evolution.

Historically, ADs used text and informal diagrams to convey the architecture. However, the diversity of stakeholder audiences and their different concerns have led to a diversity of representations of the architecture. Often, these representations are specialized based upon existing practices of the communities or disciplines involved to effectively address this variety of stakeholders and concerns (see [Software Design KA](#) and [Software Engineering Models and Methods KA](#)). These various representations are called *architecture views*.

2.1 Architecture Views and Viewpoints

An *architecture view* represents one or more aspects of an architecture to address one or more concerns [26*]. Views address distinct concerns—for example, a logical view (depicts how the system will satisfy the functional requirements); a process view (depicts how the system will use concurrency); a physical view (depicts how the system is to be deployed and distributed) and a development view (depicts how the top-level design is broken down into implementation units, the dependencies among those units and how the implementation is to be constructed). Separating concerns by view allows interested stakeholders to focus on a few things at a time and offers a means of managing the architecture’s understandability and overall complexity.

As architecture practice has evolved from the use of text and informal diagrams to the use of more rigorous representations. Each architecture view depicts architectural elements of the system using well-defined conventions, notations and models [26*]. The conventions for each view are documented as an *architecture viewpoint* [17]. Viewpoints guide the creation, interpretation and uses of architecture views. Each viewpoint links stakeholder audience concerns with a set of conventions. In model-based architecting, each view can be machine-checked against its viewpoint.

Common viewpoints include the module viewpoint, used to express a software system’s implementation in terms of its modules and their organization [2*]; the component and connector viewpoint, used to express the software’s large-scale runtime organization and interactions [2*]; the logical viewpoint, used to express fundamental concepts of the software’s domain and capability [18]; the scenarios/use cases viewpoint, used to express how users interact with the system [18]; the information viewpoint, used to express a system’s key information elements and how they are accessed and stored [26*]; and the deployment viewpoint, used to express how a system is configured and deployed for operation [26*]. Other documented viewpoints include viewpoints for availability, behavior, communications, exception handling, performance, reliability, safety and security.

Each viewpoint provides a vocabulary or language for talking about a set of concerns and the mechanisms for addressing them. The viewpoint language gives stakeholders a shared means of expression. Viewpoints need not be limited to one software system but are reusable by an organization or application community for many similar systems. When generic representations such as Unified Modeling Language (UML) are used, they can be specialized to the system, its domain or the organizations involved. (See [section 2.3 Architecture Description Languages and Architecture Frameworks](#).)

Beyond specifying forms of representation, an architecture viewpoint can capture the ways of working within a discipline or community of practice. For example, a software reliability viewpoint captures existing practices from the software reliability community for identifying and analyzing reliability issues, formulating alternatives and synthesizing and representing solutions. Like engineering handbooks, generic and specialized viewpoints provide a means to document repeatable or reusable approaches to recurring software issues. Clements *et al.* have introduced viewtypes which establish a 3-way categorization of viewpoints. These categories are module, component and connector, and allocation viewtypes [8].

Architecture descriptions frequently use *multiple* architecture views to represent the diverse structures needed to address different stakeholders' various concerns. There are two common approaches to the construction of views: the *synthetic approach* and the *projective approach*. In the synthetic approach, architects construct views of the system-of-interest and integrate these views within an architecture description using correspondence rules. In the projective approach, an architect derives each view through some routine, possibly mechanical, procedure of extraction from an underlying “uber model” [17]. A consequence of introducing multiple views into an AD is a potential mismatch between the views. Are they consistent? Are they describing the same system? This has been called the *multiple views problem* [27]. The projective approach limits possible inconsistencies, since views are derived from a single (presumably consistent) model, but at the cost of expressiveness: the underlying model may not be capable of capturing arbitrary concerns. Under the synthetic approach, architects integrate views into a whole, using linkages or other forms of traceability to cross-reference view elements to achieve consistency [17,18]. Viewpoints often include rules for establishing consistency or other relationships among views.

2.2 Architecture Styles and Patterns

Inspired by its use in the long history of the architecture of buildings, an *architectural style* is a particular manner of construction yielding a software system's characteristic features. An architectural style often expresses a software system's large-scale organization. In contrast, an *architectural pattern* expresses a common solution to a recurring problem within the context of a software system. Patterns are discussed in [section 4.4 of Software Design KA](#).

Various architectural styles and patterns have been documented [7, 27]:

- General structures (e.g., layered, call-and-return, pipes and filters, blackboard, services and microservices)
- Distributed systems (e.g., client-server, n-tier, broker, publish-subscribe, point-to-point, master-replica)
- Method-driven (e.g., object-oriented, event-driven, data flow)
- User-computer interaction (e.g., model-view-controller, presentation-abstraction-control)
- Adaptive systems (e.g., microkernel, reflection and meta-level architectures)
- Virtual machines (e.g., interpreters, rule-based, process control)

There is no strict dividing line between architectural styles and patterns. An architectural style describes the overall structure of a system or subsystem and thus defines major parts of a (sub)system and how they interact. [7, 26*] Architectural patterns exist at various ranges of scale and might be applied in a software architecture repeatedly. Both provide a solution to a particular computing problem in a given context. In fact, any architectural style can be described as an architectural pattern. [7]

In relation to architecture viewpoints, which provide the languages for talking about various aspects of software systems, a unifying notion is that both patterns and styles are *idioms* in those languages for expressing particular aspects of architectures (and designs, see [section 4.4 Design Patterns in Software Design KA](#)). An architectural pattern or style uses a vocabulary, drawn from the viewpoint's language, in a specified way, to talk about view elements, including element and relation types and their instances, and constraints on combining them [17,27]. In this way, viewpoints, patterns and styles are mechanisms for codifying recommended practices to facilitate reuse.

2.3 Architecture Description Languages and Architecture Frameworks

An *architecture description language* (ADL) is a domain-specific language for expressing software architectures. ADLs arose from module interconnection languages [25] for programming in the large. Some ADLs target a single application domain or architectural style (such as MetaH for avionics systems in an event-driven style), others are wide spectrum to frame concerns across the enterprise (such as ArchiMate™). UML has frequently been used as an ADL. ADLs often provide capabilities beyond description to enable architecture analysis or code generation.

An *architecture framework* captures the “conventions, principles and practices for the description of architectures established within a specific domain of application and/or community of stakeholders” [17]. Frameworks codify recommended practices within a specific domain and are implemented as an interlocking set of viewpoints or ADLs. Examples are OMG Unified Architecture Framework (UAF®) and ISO Reference Model for Open Distributed Processing (RM-ODP).

2.4 Architecture as Significant Decisions

Architectural design is a creative process. During this activity, architects make many decisions that profoundly affect the architecture, the downstream development process and the software system. Many factors affect decision-making, including prominent concerns of stakeholders for the software system, its requirements, and the available resources during development and throughout the life cycle. The impact on quality attributes and trade-offs among competing quality attributes are often the basis for design decisions.

The architectural design activity produces a network of decisions as its outcome, with some decisions deriving from prior decisions. Decision analysis provides one approach to architecture evaluation. Decisions should be explicitly documented, along with an explanation of the rationale for each nontrivial decision.

Architecture rationale captures *why* an architectural decision was made. This includes assumptions made before the decision, alternatives considered, and trade-offs or criteria used to select an approach and reject others. Recording rejected decisions and the reasons for their rejection can also be useful. In the future, this could either prevent a software development from making a poor decision—one rejected earlier for forgotten reasons—or allow the development to recognize that relevant conditions have changed and that they can revisit the decision.

Architectural technical debt has been introduced to reflect that today's decisions for an architecture may have significant consequences later in the software system's life cycle. Decisions deferred can compromise its maintainability or the future evolvability, and that debt will have to be paid—typically by others, not necessarily by those who caused the debt. Such debt has an economic impact on the system's future development and operations. For example, when a software project is pressed for time and designs an initial system with little modularity for its first release. The lack of modularity affects the development time for subsequent releases, impacts other developers, and perhaps the future maintainability of the system. Additional functionality can be added later only by doing extensive refactoring which impacts future timelines and introduces additional defects. [19]. Architectural technical debt can be analyzed and managed, like other concerns, using models and viewpoints [20].

3 SOFTWARE ARCHITECTURE PROCESS

This section outlines a general model of an architectural design process. It is used to demonstrate how architectural design fits into the general context of software engineering processes (see [Software Engineering Process KA](#)) and as a framework for understanding the many architecture methods currently in use. It also recognizes that architectural design can take place in a variety of contexts.

3.1 Architecture in Context

Architecture occurs in several contexts. In the traditional life cycle, there is an architectural design stage driven by software system requirements (see [Software Requirements KA](#)). Some requirements will be *architectural drivers*, influencing major decisions about the architecture, while other requirements are

deferred to subsequent stages of the software process, such as design or construction.

In product line or product family settings, a product line/family architecture is developed against a basic set of needs, requirements and other factors. That architecture will be the starting point for one or more product instances developed against specific product requirements, building upon the product baseline.

In agile approaches, there is not usually an architecture design stage. The only architecture description might be the code itself. In some agile practices, the software architecture is said to “emerge” from coding the system based on user stories through a rapid series of development cycles. Although this approach has had some success with user-centric information systems, it is difficult to ensure an adequate architecture *emerges* for other classes of applications, such as embedded and cyber-physical systems, when critical architectural properties might not be articulated by any user stories.

In enterprise and system-of-systems contexts, as in product lines and families, the overarching architecture (of the enterprise, system or product line/family) provides primary requirements and guidance on the form and constraints upon the software architecture. This baseline can be enforced through specifications, additional requirements, application programming interfaces (APIs) or conformance suites.

3.1.1 Relation of Architecture to Design

Design and architecture are often blurred. It has sometimes been said that architecture is the set of decisions that one cannot trust to designers. In fact, architecture emerged out of software design as the discipline matured, largely since the 1990s. There are various contrasts: design often focuses on an established set of requirements, whereas architecture often must shape the requirements through negotiation with stakeholders and requirements analysis. In addition, architecture often must recognize and address a wider range of concerns that may or may not end up as requirements on the software system of interest.

3.2 Architectural Design

Architectural design is the application of design principles and methods within a process to create and document a software architecture. There are many architecture methods for carrying out this activity. This section describes a general model of architectural design underlying various architecture methods based upon [14].

Architectural design involves identifying a system's major components; their responsibilities, properties, and interfaces; and the relationships and interactions among them and with the environment. In architectural design, fundamentals of the system are decided, but other aspects, such as the internal details of major components are deferred.

Typical concerns in architectural design include the following:

- Overall architecture styles and computing paradigms
- Large-scale refinement of the system into key components

- Communication and interaction among components
- Allocation of concerns and design responsibilities to components
- Component interfaces
- Understanding and analysis of scaling and performance properties, resource consumption properties, and reliability properties
- Large-scale/system-wide approaches to dominating concerns (such as safety and security, where applicable)

An overview of architectural design is presented in Fig. 3.

Architectural design is iterative, comprising three major activities: analysis, synthesis and evaluation. Often, all three major activities are performed concurrently at various levels of granularity.

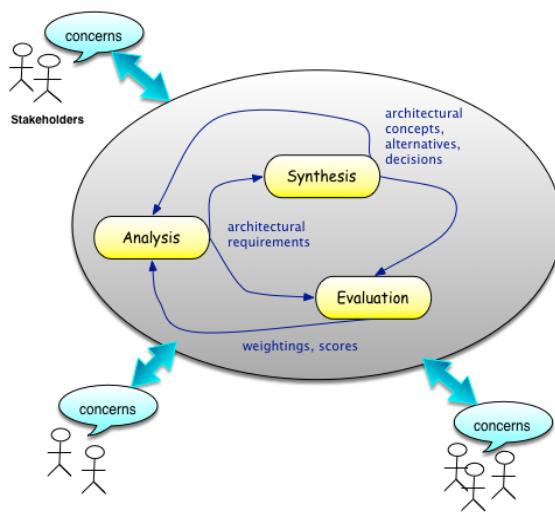


Fig. 3. A general model of architectural design

3.2.1 Architecture Analysis

Architecture analysis gathers and formulates architecture requirements (sometimes referred to as “architecturally significant requirements” or ASRs): any “requirement upon a software system which influences its architecture” [22]. Architecture analysis is based on identified concerns and on understanding the software’s context, including known requirements, stakeholder needs and the environment’s

constraints. ASRs reflect the design problems the architecture must solve. Often the combination of initial requirements and known constraints cannot be satisfied without consequences to cost, schedule, etc. In such cases, negotiation is used to modify incoming needs, requirements and expectations to make solutions possible. Architecture analysis produces ASRs, initial system-wide decisions and any overarching system principles derived from the context (see [Architecture in Context](#)).

3.2.2 Architecture Synthesis

Architecture synthesis develops candidate solutions in response to the outcomes of architecture analysis. Synthesis proceeds by working out detailed solutions to design problems identified by ASRs, and makes trade-offs to accommodate interactions between those solutions. These outcomes feedback to architecture analysis resulting in elaborated ASRs, principles and decisions which then lead to further detailed solution elements.

3.2.3 Architecture Evaluation

Architecture evaluation validates whether the chosen solutions satisfy ASRs and when and where rework is needed. Architecture evaluation methods are discussed in [topic 4 Software Architecture Evaluation](#).

3.3 Architecture Practices, Methods and Tactics

There are a number of documented architecture methods (see [Further Reading](#) for a list).

3.4 Architecting in the Large

Architectural design denotes as specific stage of the life cycle, but is only one part of software architecting. Software architecting does not occur in a vacuum, as noted in [section 3.1 Architecture in Context](#), but in an environment that often includes other architectures. For example, an application architecture should conform to an enterprise architecture; to “play well” in a system of systems, the architecture of each constituent system should conform to the system of systems architecture. In such cases, these relations need to be reflected as ASRs on the software being architected. Many software architecting activities and principles are not limited to software but equally apply to systems and enterprise architecting [21]. Weinreich and Buchgeher have extended Hofmeister et al.’s model used in [section 3.2 Architectural Design](#) to include these activities [29]:

- *architecture implementation:* overseeing implementation and certifying that implementations conform to the architecture
- *architecture maintenance:* managing and extending the architecture following its implementation
- *architecture management:* managing an organization’s portfolio of interrelated architectures
- *architecture knowledge management:* extracting, maintaining, sharing and exploiting reusable architecture assets, including decisions, lessons learned, specifications and documentation across the organization

4 SOFTWARE ARCHITECTURE EVALUATION

4.1 Goodness in Architecture

Architecture analysis takes place throughout the process of creating and sustaining an architecture. *Architecture evaluation* is typically undertaken by third parties at determined milestones as a form of assessment.

Given the multi-concern, multi-disciplinary nature of software architecture, there are many aspects to what makes an architecture “good.” The Roman architect Vitruvius posited that all buildings should have the attributes of *firmitas*, *utilitas* and *venustas* (translated from Latin as strength, utility and beauty).

Of a software system and its architecture, one can ask:

- Is it robust over its lifetime and possible evolution?
- Is it fit for its intended use?
- Is it feasible and cost-effective to construct software systems using this architecture?
- Is it, if not beautiful, then at least clear and understandable to those who must construct, use and maintain the software?

Each architecture concern may be a basis for evaluation. Evaluation is conducted against requirements (when available) or against need, expectations and norms (in other situations). A “good” architecture should address not only the distinct concerns of its stakeholders, but also the consequences of their interactions. For example: a secure architecture may be excessively costly to build and verify; an easy-to-build architecture may not be maintainable over the system’s lifetime if it cannot incorporate new technologies. The SARA Report provides a general framework for software architecture evaluation [22].

4.2 Reasoning about Architectures

Each architecture concern has a distinct basis for evaluation. Evaluation is most effective when it is based upon robust, existing architecture descriptions. ADs can be queried, examined and analyzed. For example, evaluation of functionality or behavior benefits from having an explicit architecture view or other representation of that aspect of the system to study. Specialized concerns such as reliability, safety and security often rely on specialized representations from the respective discipline.

MATRIX: TOPICS VS. REFERENCE MATERIAL

cX refers to chapter X

	Bass et al. [2*]	Budgen [6*]	Rozanski Woods [26*]	Sommerville [28*]	See also
Software Architecture Fundamentals			c2		
The senses of “architecture”	c1				[21]

Often architecture documentation is unfinished, incomplete, out of date or nonexistent. In such cases, the evaluation effort must rely on the knowledge of participants as a primary information source.

Use cases are frequently used to check an architecture’s completeness and consistency (see [Software Engineering Models and Methods KA](#)) by comparing the steps in the use case to the software architecture elements that would be involved in carrying out those steps [17].

For a general framework for reasoning about various concerns, see Bass et al. [3].

4.3 Architecture Reviews

Architecture reviews are an effective approach to assess an architecture’s status and quality and identify risks by assessing one or more architecture concerns [1]. Many reviews are informal or expertise-based, and some are more structured, organized around a checklist of topics to cover. Parnas and Weiss proposed an effective approach to conducting reviews, called *active reviews* [24], where instead of checklists, each evaluation item entails a specific activity by a reviewer to obtain the needed information.

Many organizations have institutionalized architecture review practices. For example, an industry group developed a framework for defining, conducting and documenting architecture reviews and their outcomes [22].

4.4 Architecture Metrics

An *architecture metric* is a quantitative measure of a characteristic of an architecture. Various architecture metrics have been defined. Many of these originated as design or code metrics that have been “lifted” to apply to architecture. Metrics include component dependency, cyclicity and cyclomatic complexity, internal module complexity, module coupling and cohesion, levels of nesting, and compliance with the use of patterns, styles and (required) APIs.

In continuous development paradigms (such as DevOps), other metrics have evolved that focus not on the architecture directly but on the responsiveness of the process, such as metrics for lead time for changes, deployment frequency, mean time to restore service, and change failure rate—as indicative of the state of the architecture.

Stakeholders and Concerns	c3-14		c8, c9		[17]
Roles of Architecture	c24		c30		
Architecture Description	c22		all	c6	[17]
Architecture Views and Viewpoints		c7	c3, c12, c13	c6.2	
Architectural Styles and Patterns		c6	c11	c6.3	[7]
Architecture Description Languages and Architecture Frameworks					[17]
Architecture as Significant Decisions			c8	c6.1	[17]
Architecture Processes			c7		
Architecture in Context					[21]
Architectural Design	c19-20				[14]
Architecture Methods and Tactics					see Further Reading
Architecting in the Large					[21]
Architecture Evaluation	c21		c14		[22,24]
Goodness in Architecture	c2				[3]
Reasoning about Architectures			c10		[3]
Architecture Reviews	c21				[1,22]
Architecture Metrics	c23				

FURTHER READINGS

Bass et al., Software Architecture in Practice [2*]

This book introduces concepts and recommended practices of software architecture, meaning how software is structured and how the software's components interact. The book addresses several quality concerns in detail, including: availability, deployability, energy efficiency, modifiability, performance, testability and usability. The authors offer recommended practices focusing on architectural design, architecture description, architecture evaluation and managing architecture technical debt. They also emphasize the importance of the business context in which large software is designed. In doing so, they present software architecture in a real-world setting, reflecting both the opportunities and constraints that organizations encounter.

Kruchten, The 4+1 View Model of Architecture [17].

This seminal paper organizes an approach to architecture description using five architecture viewpoints. The first four are used to produce the logical view, the development view, the process view, and the physical view. These are integrated through selected use cases or scenarios to illustrate the architecture. Hence, the model results in 4+1 views. The views are used to describe the software as envisioned by different stakeholders—such as end-users, developers, and project managers.

Rozanski and Woods, Software Systems Architecture [24*]

This is a handbook for the software systems architect. It develops key concepts of stakeholder, concern, architecture description, architecture viewpoint and architecture view, architecture patterns and styles, with examples. It provides an end-to-end architecting process. The authors provide a catalog of ready-to-use, practical viewpoints for the architect to employ that are applicable to a wide range of systems. The book is filled with guidance for applying these concepts and methods.

Clements

This book provides detailed guidance on capturing software architectures, using guidance and examples to express an architecture so that stakeholders can build, use, and maintain that system. The book introduces a 3-way categorization of views and therefore viewpoints : into module, component and connector and allocation called viewtypes, providing numerous examples of each.

Brown, Software Architecture for Developers [5]

Brown provides an overview of software architecture topics from the perspective of a developer. He discusses common architecture drivers (functional requirements, quality concerns, constraints and architecture principles. He has an in-depth discussion of the role of the architect in a development setting and requisite knowledge and skills for architects. He focuses on the practical issues of architecture in the delivery

process and on managing risk. An appendix provides a case study.

Fairbanks, Just Enough Software Architecture: A risk-driven approach [12]

Fairbanks offers a risk-driven approach to architecting within the context of development: do just enough software architecture to mitigate the identified risks where those risks could result from a small solution space, from extremely demanding quality requirements or from possible high-risk failures. The risk-driven approach is harmonious with low-ceremony and agile approaches. Architecting, as argued by Fairbanks, is not just for architects—but is relevant to all developers.

Erder, Pureur and Woods, Continuous Architecture in Practice: Software Architecture in the Age of Agility and DevOps. [11]

This book shows how “classical” thinking about software architecture has evolved in the present day in the contexts of agile, cloud-based and DevOps approaches to software development by providing practical guidance on a range of quality and cross-cutting concerns including security, resilience, scalability and integration of emerging technologies.

REFERENCES

- [1] M. Ali Babar, and I. Gorton, “Software Architecture Review: The State of the Practice”, *IEEE Computer*, July 2009.
- [2] * L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 4th edition, 2021.
- [3] L. Bass, J. Ivers, M.H. Klein, and P. Merson, Reasoning Frameworks, CMU/SEI-2005-TR-007, 2005.
- [4] * F. Brooks, *The Design of Design*, Addison-Wesley, 2010.
- [5] S. Brown, Software Architecture for Developers, 2018, <http://leanpub.com/software-architecture-for-developers>
- [6] * D. Budgen, Software Design: Creating Solutions for Ill-Structured Problems, 3rd Edition, CRC Press, 2021.
- [7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern Oriented Software Architecture*, John Wiley & Sons, 1996.
- [8] P. Clements, et al. Documenting Software Architecture: Views and Beyond, 2nd edition Addison Welsley, 2011
- [9] M.E. Conway, “How Do Committees Invent?” Datamation, 14(4), 28-31, 1968.
- [10] E.W. Dijkstra, “On the role of scientific thought”, 1974, available at http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD_447.html.
- [11] P. Eeles, and P. Cripps, *The Process of Software Architecting*, Addison Wesley, 2010.
- [12] M. Erder, P. Pureur and E. Woods, *Continuous Architecture in Practice: Software Architecture in the Age of Agility and DevOps*, Addison-Wesley, 2021.
- [13] G. Fairbanks, *Just Enough Software Architecture: A Risk-Driven Approach*, Marshall & Brainerd, 2010.
- [14] C. Hofmeister, P.B. Kruchten, R.L. Nord, H. Obbink, A. Ran, and P. America, “A general model of software architecture design derived from five industrial approaches”, *The Journal of Systems and Software*, 80, 106–126, 2007.
- [15] C. Hofmeister, R.L. Nord, and D. Soni, *Applied Software Architecture*, Addison- Wesley, 2000.
- [16] ISO/IEC/IEEE 24765:2017, Systems and software engineering — Vocabulary.

- [17] ISO/IEC/IEEE 42010:2011, Systems and software engineering — Architecture description.
- [18] P.B. Kruchten, The “4+1” View Model of Architecture, IEEE Software 12(6), 1995.
- [19] P.B. Kruchten, Nord, R.L.; Ozkaya, I.: *Managing Technical Debt: Reducing Friction in Software Development*. Addison Wesley, 2019.
- [20] Z. Li, P. Liang, P. Avgeriou, Architecture viewpoints for documenting architectural technical debt. *Software Quality Assurance*, Elsevier, 2016.
- [21] * M.W. Maier, and E. Rechtin, *The Art of Systems Architecting*, 3rd edition, CRC Press, 2021.
- [22] H. Obbink, et al., *Report on Software Architecture Review and Assessment* (SARA), version 1.0, available at <https://philippe.kruchten.com/architecture/SARAv1.pdf>, 2002.
- [23] D.L. Parnas, “On the criteria to be used in decomposing systems into modules”, Communications of the ACM 15(12), 1053-1058, 1972.
- [24] D.L. Parnas, and D.M. Weiss, “Active Design Reviews: Principles and Practices”, Proceedings of 8th International Conference on Software Engineering, 215-222, 1985.
- [25] R. Prieto-Diaz, and J.M. Neighbors, “Module Interconnection Languages”, Journal of Systems and Software, 6(4), 307–334, 1986.
- [26] * N. Rozanski, and E. Woods, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*, 2nd edition, Addison-Wesley, 2011.
- [27] M. Shaw, and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
- [28] * I. Sommerville, *Software Engineering*, 10th edition, 2016.
- [29] R. Weinreich, and G. Buchgeher, Towards supporting the software architecture life cycle, *The Journal of Systems and Software*, 85, 546–561, 2012.

chapter 3

Software Design

ACRONYMS

API	application programming interface
AOD	aspect-oriented design
CBD	component-based design
CRC	class responsibility collaborator (or collaboration)
DFD	data flow diagram
DSL	domain-specific language
ERD	entity relationship diagram
FOSS	free and open source software
IDL	interface description language
MBD	model-based design
MDD	model-driven design
OO	object-oriented
PDL	program design language
SDD	software design description
UML	Unified Modeling Language

INTRODUCTION

This chapter considers software design from several perspectives—focusing on basic concepts, context and processes, software design qualities and strategies, and recording and evaluating designs.

Design is used in distinct but closely related ways to refer to (1) the *discipline* (“use of scientific principles, technical information, and imagination in the definition of a software system to perform [prespecified] functions with maximum economy and efficiency”) [10]; (2) the *processes* for performing within that discipline; (3) the *result* of applying that discipline; and (4) the *stage* in the life cycle of a software system during which those processes yield those results.

Software design, viewed as a life cycle activity, is the application of software engineering discipline in which software requirements are analyzed to define the software’s external characteristics and internal structure as the basis for the software’s construction.

A *software design description* (SDD) documents the result of software design. It is a “representation of software created to facilitate analysis, planning, implementation, and decision-making. The software design description is used as a medium for communicating software design information and can be thought of as a blueprint or model of the system” [10].

The SDD, which may take many forms, encompasses the refinement of that software into components, the organization of those components, and the definition of interfaces among them and between the software and the outside world—to a level of detail that enables their construction.

Software design takes place in three stages:

- architectural design of the software system
- high-level or external-facing design of the system and its components
- detailed or internal-facing design

Architectural design is a part of architecting, discussed in the [Software Architecture KA](#).

BREAKDOWN OF TOPICS FOR SOFTWARE DESIGN

The breakdown of topics for the Software Design KA is shown in Fig. 1.

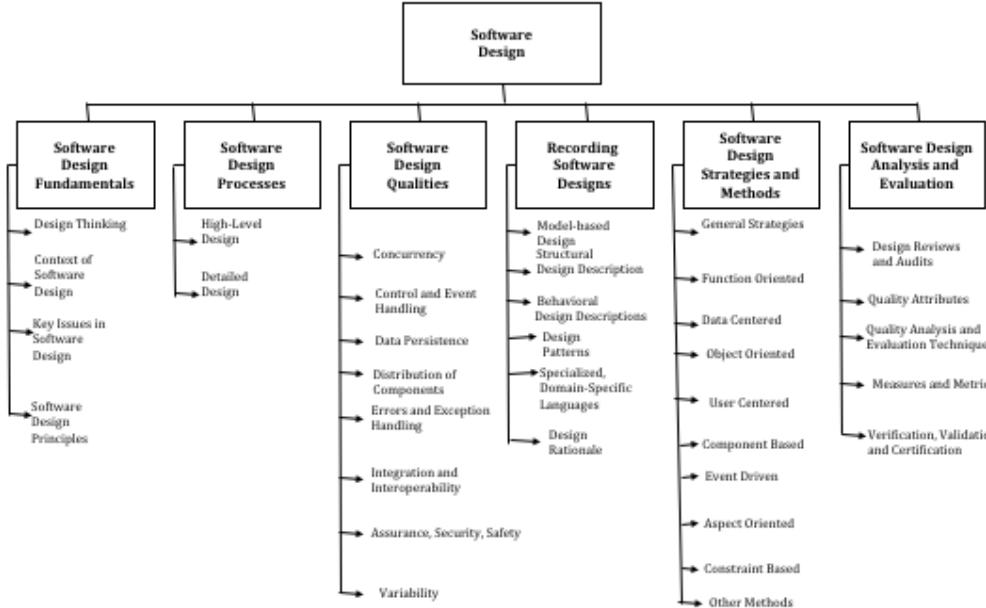


Fig. 1. Breakdown of topics for the Software Design KA

1 SOFTWARE DESIGN FUNDAMENTALS

The concepts, notions and terminology introduced here form a basis for understanding the role and scope of software design.

1.1 Design Thinking

Design is all around us, in the things and organizations that have been created to meet a need or solve a problem.

In a general sense, design can be viewed as a form of problem-solving. For example, the concept of a *wicked problem*—a problem with no definitive solution—is interesting in terms of understanding the limits of design. Many other notions and concepts help us understand design in its general sense: goals, constraints, alternatives, representations and solutions. (See also [Problem-Solving Techniques in the Computing Foundations KA](#).)

Design thinking comprises two essentials: (1) understanding the need or problem and (2) devising a solution. Ross, Goodenough and Irvine offer an elaboration of design thinking appropriate to software:

This process consists of five basic steps: (1) crystallize a purpose or objective; (2) formulate a concept for how the purpose can be achieved; (3) devise a mechanism that implements the conceptual structure; (4) introduce a notation for expressing the capabilities of the mechanism and invoking its use; (5) describe the usage of the notation in a specific problem context to invoke the mechanism so the purpose is achieved. [18]

This is particularly appropriate because much of software design consists of creating the necessary *vocabulary* to express

a problem, express its solution and implement that solution. The steps emphasize the linguistic nature of software design problem solving. This is a recurring pattern we see throughout high-level design, detailed design and architecting (see [Architecting in the Large in Software Architecture KA](#)). Therefore, Software Design is a practical process of transforming a problem statement into a solution statement. Software design shares commonalities with other kinds of design. Design can be further understood via *design theory* [7].

1.2 Context of Software Design

Software design is an important part of the software development process. To understand the role of software design is to see how it fits into the software development life cycle (see [Software Process KA](#)). To understand that context, it is important to understand the major characteristics and roles of software requirements, software construction, software testing, and software maintenance. The context varies with many factors, including degree of formality and stage of the life cycle.

- Design interface with software requirements: The requirements establish a set of problems that the software design must solve.
- Design interface with software architecture: In cases where an architecture has been established, that architecture constrains the design by capturing fundamental aspects of the system: its major components and their interconnections, application programming interfaces (APIs), styles and patterns to be used, and architectural principles to be observed and enforced.

- Design interface with software construction: The software design must provide a guide to implementors on building the system.
- Design interface with software testing: Software design provides a foundation for an overall testing strategy and test cases that ensure that the design is properly implemented and operates as intended.

1.3 Key Issues in Software Design

Many key issues must be dealt with when designing software. Some are quality *concerns* that all software must address (performance, security, reliability, usability, etc.). Another important issue is how to refine, organize, interconnect and package software components. These issues are so fundamental that all design approaches address them in one way or another. (See [topic Stakeholders and Concerns in Software Architecture KA](#), section 1.4 *Software Design Principles*, and [topic 5 Software Design Strategies and Methods](#).)

In contrast, other issues “deal with some aspect of software’s behavior that is not in the application domain, but which addresses some of the supporting domains” [2]. Such issues, which often crosscut the system’s functionality, are referred to as *aspects*, which “tend not to be units of software’s functional decomposition, but rather to be properties that affect the performance or semantics of the components in systemic ways” [11].

1.4 Software Design Principles

A *principle* is “a fundamental truth or proposition that serves as the foundation for a system of belief or behavior or for a chain of reasoning.” [[Oxford English Dictionary](#)]

Design principles provide direction or guidance for making decisions during design. Some principles originated during the early days of software engineering, others even predate the discipline, deriving from best practices in engineering unrelated to software. (See [Engineering Foundations KA](#).) Decision making can also be assisted by quantitative methods, such as discussed in [Software Engineering Economics KA](#). Software design principles are key notions that provide the basis for many different software design concepts, approaches and methods. The principles listed below apply to any of the three stages of design. Many of these principles are interrelated. Whether alone or used in combination with other principles, they are reflected elsewhere in software design to produce many concepts and constructs found in design capture, strategies and methods. This is itself an application of the design thinking process above. Software design principles include the following:

- *Abstraction* is “a view of an object that focuses on the information relevant to a particular purpose and ignores the remainder of the information” [10]. “The abstraction principle . . . helps to identify essential properties common to superficially different entities” [18]. (See also topic [Abstraction in the Computing Foundations KA](#).)

- *Separation of concerns (SoC)*. A *design concern* is an “area of interest with respect to a software design” [10] that is relevant to one or more of its stakeholders. By identifying and separating concerns, the designer can focus on each concern for the system in isolation about which Dijkstra says “even if not perfectly

possible, [SoC] is yet the only available technique for effective ordering of one’s thoughts” [5] (See also [topic Stakeholders and Concerns in Software Architecture KA](#).)

- *Modularization* (or *refinement* or *decomposition*) structures large software as comprising smaller components or units. Each component is named and has well-defined interfaces for its interactions with other components. Smaller components are easier to understand and, therefore, to maintain. There are numerous modularization strategies. (See [topic 5 Software Design Strategies and Methods](#).)

Traditionally, the goal is to place distinct functionalities and responsibilities in different components. Parnas advocated that each module in a system should have a single responsibility [15]. One way to think of modularization is as a special case of more general strategies, such as separation of concerns or divide and conquer. (see [topic Problem-Solving Techniques in Computing Foundations](#)).

- *Encapsulation* (or *information hiding*) builds upon the principles of abstraction and modularization so that nonessential information is less accessible, allowing users of the module to focus on the essential elements of the interface.

- *Separation of interface and implementation* is an application of encapsulation that involves defining a component by specifying its public interfaces, which are known to and accessible to clients; isolating the use of a component from the details of how that component is built. (See [Encapsulation \(or information hiding\)](#) above.)

- *Coupling* is defined as “a measure of the interdependence among modules in a computer program” [10]. Most design methods advocate that modules should be loosely or weakly coupled.

- *Cohesion* (or *localization*) is defined as “a measure of the strength of association of the elements within a module” [10]. Cohesion highlights organizing a module’s constituents based on their relatedness. Most design methods advocate that modules should maximize their cohesion/locality.

- *Uniformity* is a principle of consistency across software components—common solutions should be produced to address common or recurring problems. These include naming schemes, notations and syntax, interfaces that define access to services and mechanisms, and ordering of elements and parameters. This can be achieved through conventions such as rules, formats and styles.

- *Completeness* (or *sufficiency*) means ensuring that a software component captures the important characteristics of an abstraction and leaves nothing out. Completeness takes various forms, perhaps the most important of which is design completeness against requirements: a design should be sufficient for designers to demonstrate how requirements will be met and how subsequent work will satisfy those requirements. Design should be complete with respect to the modes and states of the software.

- *Confirmability* means that information needed to verify that the software is correct, complete and fit for use is available. This is relevant for any software but is of particular importance for high-assurance software, such as software where security,

reliability or safety-critical concerns are present. An SDD should be sufficient as a basis for verifying a design.

- *Other design principles.* Recently, with the increased appearance of autonomous systems, the use of machine learning and artificial intelligence, and, generally, systems with widening social impacts, approaches to *Ethically Aligned Design* have been developed to address concerns including universal human values, political self-determination, and data agency and technical dependability [8]. The general principles of Ethically Aligned Design are human rights, well-being, data agency, effectiveness, transparency, accountability, awareness of misuse, and competence.

2 SOFTWARE DESIGN PROCESSES

Software design is generally considered a multistage process or activity. Software design can be divided into the following stages or phases. When necessary, we distinguish the phase from the general activity:

- Architectural design stage
- High-level design stage
- Detailed design stage

The architectural design stage addresses the fundamentals of the system as a whole and situated in its environment (see [Software Architecture KA](#)).

The high-level design stage is *outward-facing*—developing the top-level structure and organization of the software, identifying its various components and how that software system and its components interact with the environment and its elements.

The detailed design stage is *inward-facing*—specifying each component in sufficient detail to facilitate its construction and to meet its outside obligations, including how software components are further refined into modules and units.

Each stage reflects the basic pattern outlined in section 1.1 [Design Thinking](#).

Not all stages are found in every software process. However, when present, each stage creates an obligation upon the next stage regarding the software which is under development.

Although software developers generally follow similar guidelines for what happens in each stage, there are no strict boundaries between stages regarding what must be done and when. For example, for many software systems, the choice of an algorithm to sort data will be deferred to programmers, within the constraints and guidance provided by the system's requirements, its architecture description or design specifications. However, for another software system, the existence of a suitable algorithm could be architecturally significant and must be determined early in the life cycle. Without that algorithm, there is no possibility of constructing the software to meet its requirements.

Some rules of thumb for each stage include the following:

- The architectural design stage defines a computational model, the major computational elements, and the

important protocols and relationships among them. This stage develops strategies to address crosscutting concerns, such as performance, reliability, security and safety, and articulation of crosscutting decisions, including system-wide styles (e.g., a Model-View-Controller style versus, a Pipes-and-Filters style, together with the rationale for such decisions).

- The high-level design stage includes identification of the primary computational elements and significant relationships among them, with a focus on each major component's existence, role and interfaces. That definition should be sufficiently detailed to allow designers or programmers of client components to correctly and efficiently access each server's capabilities—without having to read its code.
- The detailed design stage defines each module's internal structure, focusing on detailing and justifying choices of algorithms, data access and data representation. The detailed design specifications should be sufficient to allow programmers to code each module during construction (see [Software Construction KA](#)). The code is a representation of the solution that is sufficiently detailed and complete that a compiler (or interpreter) can execute it.

2.1 High-Level Design

High-level design specifies the interaction of a system's major components with one another and with the environment, including users, devices and other systems. High-level design addresses the following:

- External events and messages to which the system must respond
- Events and messages which the system must produce
- Specification of the data formats and protocols for events and messages
- Specification of the ordering and timing relationships between input events and messages, and output events and messages
- Data persistence (how data is stored and managed)

High-level design is undertaken within the envelope established by the system's software architecture (if any). Each of the above may be guided or constrained by architecture directives. For example, event signaling and messaging will use the protocols and modes of interaction established by the architecture. Data formats and protocols will use data and communication standards specified by the architecture. Absent an explicit architecture design stage, some of these directives will be established by the software requirements or decided during high-level design.

2.2 Detailed Design

The detailed design stage proceeds within the constraints established by the high-level design. It specifies major system components' internal characteristics, internal modules and their interconnections to other modules, services and processes they

provide, computing properties, algorithms, and data access rules and data structures. This includes the following:

- Refinement of major system components into modules or program units, including opportunities for using off-the-shelf components and application frameworks
- Allocation of design responsibilities to modules and program units
- Interactions among modules
- Scope and visibility among components, modules and program units
- Component modes, component states and transitions among them
- Data and control interdependencies
- Data organization, packaging and implementation
- User interfaces
- Requisite algorithms and data structures

3 SOFTWARE DESIGN QUALITIES

Software requirements and architecture directives are intended to guide software toward certain characteristics or *design qualities*. Design qualities are an important subclass of concerns (see topic [Stakeholders and Concerns in Software Architecture KA](#)). One role of design principles (see section [1.4 Software Design Principles](#)) is to help software achieve these qualities. Among the characteristics of interest to designers are the following:

3.1 Concurrency

Design for concurrency concerns how software is refined into concurrent units such as processes, tasks, and threads and the consequences of those decisions with respect to efficiency, atomicity, synchronization and scheduling.

3.2 Control and Event Handling

Event handling is concerned with how to organize control flow as well as how to handle reactive and temporal events through various mechanisms including synchronization, implicit invocation and callbacks.

3.3 Data Persistence

Data persistence concerns the storage and management of data throughout the system.

3.4 Distribution of Components

Distribution concerns how software components are distributed across hardware (including computers, networks and other devices) and how those components communicate while meeting performance, reliability, scalability, availability, monitorability, business continuity and other expectations.

3.5 Errors and Exception Handling, Fault Tolerance

This concern pertains to how to prevent, avoid, mitigate, tolerate and process errors and exceptional conditions.

3.6 Integration and Interoperability

This issue arises at the enterprise or system-of-systems level or for any complex software when heterogeneous systems or applications need to interwork through exchanges of data or accessing one another's services. Within a software system, the issue arises when components are designed using different frameworks, libraries or protocols.

3.7 Assurance, Security and Safety

High assurance spans a number of software qualities, including security and safety concerns, pertaining to whether the software behaves as intended in critical situations, such as in the face of hazards. Design for security concerns how to prevent unauthorized disclosure, creation, change, deletion, or denial of access to information and other resources in the face of attacks upon the system or violations of system policies to limit damage; provide continuity of service; and assist repair and recovery. Design for safety pertains to managing the software's behavior in circumstances which might lead to harm to or loss of human life or damage to property or the environment.

3.8 Variability

Variability concerns permissible variations in software, as arises for example in software product lines and system families, to accommodate and manage multiple different deployments such as for different organizations or markets.

4 RECORDING SOFTWARE DESIGNS

The outcome of design processes is accumulated knowledge and work products recording that knowledge. Work products of software design capture (1) aspects of the problems to be solved, using the vocabulary of the domain; (2) a solution vocabulary for solving the design problems (see section [1.1 Design Thinking](#)); (3) the major decisions that have been taken; and (4) explanations of the rationale for each nontrivial decision. Recording the rationale for important decisions enhances the software product's long-term maintainability when modifications or enhancements are considered (see section [4.6 Design Rationale](#)). These work products, often termed *design descriptions* or *design specifications*, can take the form of texts, diagrams, models and prototypes that comprise the blueprints of the software to be implemented.

A fundamental aspect of software design is communication about the design to customers, other designers, implementers and other stakeholders. This is true whether the software is developed using agile, traditional or formal methods. The communication will vary depending upon the target audience, the level of detail being communicated, and relevance to the concerns of the stakeholders. While implementers are an important audience for design specifications, testing and quality assurance personnel, certification authorities, and requirements analysts will also use the design specifications in their work. Therefore, design specifications should have a clearly defined audience, subject and intended usage.

Designers can analyze and evaluate these work products to determine whether the design can meet the requirements and constraints on the software. Software design also examines and evaluates alternative solutions and trade-offs. In addition to using them as inputs and as the starting point for construction

and testing, stakeholders can use the design work products to plan subsequent activities, such as system verification and validation.

As design concepts evolve, so do their representations (see section 1.1 *Design Thinking*); part of the design process involves creating appropriate vocabularies for problems and solutions. An informal sketch may be most appropriate for the early stages. It is useful to distinguish in-process (“working”) specifications from final design products. The former are produced by the design team for the design team; the latter may be produced for known stakeholders or even for an unknown future audience.

Many notations exist to represent software design artifacts. Software design is often carried out using multiple types of notation. Two broad areas of concern are software structures and software behaviors. Some are used to describe a design’s structural organization, others to represent the software’s intended behavior. Below, they are categorized as notations for structural and behavioral concerns (see section 4.2 *Structural Design Descriptions* and section 4.3 *Behavioral Design Descriptions*, respectively). Certain notations are used mostly during architectural design and others mainly during detailed design; some are useful throughout all stages of software design. Some notations are closely tied to the context of specific design methods (see [Software Design Strategies and Methods KA](#)).

The Unified Modeling Language (UML) is a widely used family of notations addressing both structural and behavioral concerns and is used in all design stages, from architectural through detailed design [1].

4.1 Model-Based Design

Over the history of software engineering, including architecture and design, there has been an evolution from document-based artifacts to model-based artifacts. *Model-based design* (MBD) is an approach to recording designs where models play an important role.

This trend reflects the limitations of document-based artifacts and the increased capabilities of automated tools. Document-based artifacts use natural language and informal diagrams to convey designers’ intentions, which might introduce ambiguity and incompleteness. Even when documents use well-defined formats, relevant information might be spread across documents, making understandability and analysis difficult. With MBD, appropriate tooling can gather and organize relevant information for use by designers and other stakeholders in an accessible form.

Modern tools have accelerated the trend from document to model-based artifacts. Tooling enables animation or simulation of various software aspects, analyses of what-if scenarios and trade-offs, and rapid prototyping. Tooling also facilitates continuous testing and integration approaches, enhanced and interactive traceability, and knowledge capture and management, which are inefficient or even infeasible with document-based approaches.

Model-driven development (MDD) is a development paradigm that uses models as the development process’ primary artifacts (see [Software Engineering Models and Methods KA](#)).

4.2 Structural Design Descriptions

The following types of notation, most of which are graphical, are used to represent the structural aspects of a software design—that is, they are used to describe the major components and how they are interconnected (static view) and the allocation of responsibilities to components and modules:

- Class and object diagrams are used to represent a set of classes and objects and their interrelationships.
- Component diagrams are used to represent a set of *components* (“physical and replaceable part[s] of a system that [conform] to and [provide] the realization of a set of interfaces” [1]) and their interconnections. Component models evolved from earlier module interconnection languages into the package systems of programming languages like Ada and Java and the sophisticated module systems of current functional language systems such as Haskell and Coq.
- Class responsibility collaborator cards (CRCs) are used to denote the names of components (classes), their responsibilities and the components they interact with to meet those responsibilities.
- Deployment diagrams are used to represent a set of physical nodes and their interconnections to model the physical aspects of software as deployed on hardware.
- Entity relationship diagrams (ERDs) are used to represent conceptual, logical and physical models of data as stored in information repositories or as a part of interface descriptions.
- Interface description languages (IDLs) are programming-like languages used to define the interfaces (names and types of exported operations) of software components.
- Structure charts are used to describe the calling structure of programs (that is, they show which modules call, and are called by, which other modules).

4.3 Behavioral Design Descriptions

The following notations and languages, some graphical and some textual, are used to describe the dynamic behavior of software systems and their components. Many of these notations are useful mostly, but not exclusively, during detailed design. Moreover, behavioral descriptions can include rationale for design decisions (see section 4.6 *Design Rationale*).

- Activity diagrams are used to show flow of a computation from activity to activity. They also can represent concurrent activities, their inputs and outputs and opportunities for concurrency.
- Communication diagrams are used to show the interactions among a group of objects; the emphasis is on the objects, their links and the messages they exchange on those links.
- Data flow diagrams (DFDs) are used to show data flow among computing elements. A DFD provides “a description based on modeling the flow of information

around a network of operational elements, with each element making use of or modifying the information flowing into that element” [4]. DFDs have other uses, such as security analysis, as they identify possible paths for attack and disclosure of confidential information.

- Decision tables and diagrams are used to represent complex combinations of conditions and actions.
- Flowcharts are used to represent the flow of control and the sequence of associated actions.
- Sequence diagrams are used to show the interactions among a group of objects, depicting the time ordering of messages passed among objects.
- State (transition) diagrams and statecharts are used to show transitions from state to state and how a component’s behavior changes based on its current state and response to input events.
- Formal specification languages are predominantly textual languages founded upon basic notions from mathematics (for example, type, set, sequence, logical proposition) to rigorously and abstractly define software component interfaces and behavior, often in terms of pre- and post-conditions, invariants, type checking, and computational models (see [section Formal Methods in Software Engineering Models and Methods KA](#)).
- Pseudocode and program design languages (PDLs) are structured, programming language-like notations used to describe a procedure’s processing behavior, generally at the detailed design stage. The use of these languages is less common today but is still found in the documentation of algorithms.

4.4 Design Patterns and Styles

Succinctly described, a pattern is “a common solution to a common problem in a given context” [6]. Architectural styles can be viewed as patterns “in the large,” describing common solutions to architecture-level problems that pervade the software. Design patterns include the following:

- Creational patterns (e.g., builder, factory, prototype, singleton)
- Structural patterns (e.g., adapter, bridge, composite, decorator, façade, flyweight, proxy)
- Behavioral patterns (e.g., command, interpreter, iterator, mediator, memento, observer, peer-to-peer, publish-subscribe, state, strategy, template, visitor)

Design patterns and styles reflect idioms that have proven useful in solving particular design problems in the past. They arise at all stages of design, including architecture (see also [section 2.2 Architecture Styles and Patterns, Software Architecture KA](#)).

4.5 Specialized and Domain-specific languages

Not every design representation falls easily into the structure/behavior dichotomy. For example, user interface design mixes the structural layout of what a user might see with

the behavioral logic of sequencing screens based upon user actions. Specialized concerns such as safety and reliability often have their own forms of representation that have evolved among specialists in those communities [19].

A recent trend has been the maturing of *domain-specific languages* (DSLs) and widely available tools to develop them. In this approach, part of the design process is codifying concepts and constructs of a specific application domain to create a computer language for that domain so that representing the design using these constructs leads to an animated or executable implementation. DSLs blur the lines among modeling languages, design languages and programming languages in this approach.

There are DSLs and supporting tools for domains such as simulation; real-time, reactive and distributed systems; game development; user interfaces; test development; and language processing tools. The growth of DSLs has been facilitated by increasingly powerful grammar-driven tools that, given a language definition, can generate a graphical user interface, syntax checkers, code generators, compilers and linkers for the specialized language.

4.6 Design Rationale

A useful design outcome is insight into and explicit documentation of the major decisions taken, along with an explanation of the rationale for each decision. Design rationale captures *why* a design decision was made. This includes prior assumptions made, alternatives considered, and trade-offs and criteria analyzed to select one approach and reject others.

Although the reasons for decisions are likely to be obvious to the current design team, they can be less obvious to those who modify or maintain the system after deployment. Recording the rationale enhances the software product’s long-term maintainability. Continuing to capture the rationale for changes during maintenance also contributes to the software’s viability.

It can also be useful to capture rejected decisions and the reasons for rejection. Capturing these rationales can enable a team to revisit a previously rejected decision when assumptions, requirements or constraints change. The importance of rationale is visible, for example, in free and open-source software (FOSS) projects, which often involve large, distributed teams of developers with frequent turnover.

Design rationale may be captured as part of a software design description or as a companion artifact. Often rationale is captured in text, but other forms of representation can also be used, such as graphs that portray a design as an interconnected network of decisions.

5 SOFTWARE DESIGN STRATEGIES AND METHODS

Various strategies and methods exist to structure and guide the design process; many of these evolved from programming styles or paradigms. In addition to embodying one or more general strategies, most design methods focus on making one or more design concepts (whether objects, methods or events) prominent as organizing themes for the software. These themes then guide the designers as to what to focus on first, how to proceed, and how to structure modules.

5.1 General Strategies

Some often-cited examples of general strategies useful in the design process include divide-and-conquer and stepwise refinement strategies; top-down vs. bottom-up strategies; strategies using heuristics, patterns and pattern languages; and iterative and incremental approaches.

5.2 Function-Oriented (or Structured) Design

This is one of the classical software design methods. It focuses on refinement (or decomposition) to identify major software functions, elaborating them in a top- down manner. Structured design often follows structured analysis, producing DFDs and associated process descriptions. Various tools enable the automated translation of DFDs into high-level designs.

5.3 Data-Centered Design

Data-centered design starts from the data structures a program manipulates rather than from the functions it performs. The software designer specifies the input and output data structures and then develops program units that transform inputs into outputs. Various heuristics have been proposed to deal with special cases, such as cases where there is a mismatch between the input and output structures.

5.4 Object-Oriented Design

Numerous software design methods based on objects have been proposed. The field has evolved from the early object-oriented design of the mid-1980s (where nouns depict objects; verbs depict methods; and adjectives depict attributes), where inheritance and polymorphism play key roles, to the field of component-based design (CBD), where metainformation can be defined and accessed (through reflection, for example). Although OOD's roots stem from the concept of data abstraction, responsibility-driven design has been proposed as an alternative underlying principle of OOD.

5.5 User-Centered Design

User-centered design is more than a design method; it is a multidisciplinary approach emphasizing a deep understanding of users and their needs as the basis for designing user experiences within the context of their organization and the tasks to be accomplished. It involves gathering user requirements, creating a user flow of tasks and decisions, creating prototypes or mockups representative of user interfaces, and evaluating the design solution against original requirements [14].

5.6 Component-Based Design (CBD)

CBD decomposes a software system into one or more standalone components that communicate only on well-defined interfaces and conform to a system-wide standard component model. A software component is an independent unit, having well-defined interfaces and dependencies that can be composed and deployed independently. CBD addresses issues related to providing, developing and integrating such components to improve reuse. CBD often emphasizes common APIs for all components and specialized APIs for specific services or responsibilities.

5.7 Event-Driven Design

Event-driven design is an approach where a system or component invokes its operations in reaction to events (indirect

invocation) [13]. Publish/subscribe messaging (broadcasting) is often used as means of transporting events via the network to all interested subscribers. Publish/subscribe keeps the producers and consumers decoupled using a message broker with channels called topics. This differs from Point-to-point messaging where senders and receivers need to know each other to deliver and receive a message. Different types of event processing exist, i.e. simple event processing, event stream processing and complex event processing. Message-based systems frequently incorporate identifiable senders and receivers within the design. Event-driven systems may not identify senders and receivers explicitly—instead each module produces events while listening for any events they care about or need to respond to [12]. “Anonymous” asynchronous message and event processing are good strategies for scalable systems.

5.8 Aspect-Oriented Design (AOD)

AOD is a method by which software is constructed using aspects to implement the crosscutting concerns and extensions identified in software requirements [11]. AOD evolved from object-oriented design and programming practices. Although it has yet to become a widespread design or programming paradigm, the aspect-oriented perspective is frequently used in application frameworks and software libraries where parameters of the framework or library can be configured with aspect declarations.

5.9 Constraint-Based Design

Constraints' role in the design process is to limit the size of a design space to exclude infeasible or unacceptable alternatives. Constraints accelerate design because they force a few early decisions. The constraints can reflect limits imposed on the hardware, software, data, operational procedures, interfaces or anything that affects the software. The constrained design space can then be explored with search or backtracking methods. Constraint-based design approaches are used in user interface design, gaming and other applications. In general, constraint satisfaction problems can be NP-hard; however, various kinds of constraint-based programming can be used to approximate or solve constraint problems.

5.10 Other Methods

Other approaches to design exist (see [Software Engineering Models and Methods KA](#)). For example, iterative and adaptive methods implement software increments and reduce the emphasis on rigorous software requirements and design.

Service-oriented methods builds distributed software using web services executed on distributed computers. Software systems are often constructed using services from different providers interconnect with standard protocols (e.g., HTTP, HTTPS, SOAP) designed to support service communication and service information exchange.

6 SOFTWARE DESIGN QUALITY ANALYSIS AND EVALUATION

6.1 Design Reviews and Audits

Design reviews are intended as comprehensive examinations of a design to assess concerns such as status or degree of completion, coverage of requirements, open or unresolved issues and potential problems. A design review can be undertaken at

any stage of design. Design reviews can be conducted by the design team, by an independent third party or other stakeholder. A design audit is more narrowly focused on a set list of characteristics (e.g., a functional audit). (See also [section 2.3 Reviews and Audits in Software Quality KA](#).)

6.2 Quality Attributes

Various attributes contribute to the quality of a software design, including various “ilities” (maintainability, portability, testability, usability) and “nesses” (correctness, robustness). Qualities are a major subset concerns (see [topic Stakeholders and Concerns in Software Architecture KA](#)). Some qualities can be observed at runtime (e.g., performance, security, availability, functionality, usability); others cannot (e.g., modifiability, portability, reusability, testability); some (e.g., conceptual integrity, correctness, completeness) are intrinsic to the software.

6.3 Quality Analysis and Evaluation Techniques

Various tools and techniques can help in analyzing and evaluating software design quality. (See also [topic Software Quality Tools in Software Quality KA](#).)

- Software design reviews include informal and rigorous techniques to determine software qualities based on SDDs and other design artifacts for example, architecture reviews, design reviews and inspections; scenario-based techniques; requirements tracing.
- Static analysis: formal or semiformal static (nonexecutable) analysis that can be used to evaluate a design (for example, fault-tree analysis or automated cross-checking). Design vulnerability analysis (for example, static analysis for security weaknesses) can be performed if security is a concern. Formal design analysis uses mathematical models that allow designers to predict the behavior and validate the performance of the software instead of having to rely entirely on testing. Formal design analysis can be used to detect residual specification and design errors (perhaps caused by imprecision, ambiguity, and sometimes other kinds of

mistakes). (See also [Software Engineering Models and Methods KA](#).)

- Simulation and prototyping: dynamic techniques to evaluate a design (for example, performance simulation or feasibility prototypes).

6.4 Measures and Metrics

Measures can be used to assess or to quantitatively estimate various aspects of a software design; for example, size, structure, or quality. Most measures that have been proposed are based upon the approach used for producing the design (see [topic 5 Software Design Strategies and Methods](#)). These measures are classified in two broad categories:

- Function-based (structured) design measures: measures obtained by analyzing functional decomposition; generally represented using a structure chart (or hierarchical diagram) on which various measures can be calculated.
- Object-oriented design measures: the design structure is typically represented as a class diagram, on which various measures can be computed. Measures on the properties of the internal content of each class can also be calculated.

6.5 Verification, Validation and Certification

Systematic analysis or evaluation of the design plays an important role in each of these three areas:

- verification: to confirm that the design satisfies stated requirements;
- validation: to establish that the design will allow the system to meet the expectations of its stakeholders, including customers, users, operators and maintainers;
- certification: third-party attestation of conformity of design to its overall specification and intended usage.

(See also [section 2.2 Verification and Validation in Software Quality KA](#).)

MATRIX: TOPICS VS. REFERENCE MATERIAL

In table below, cX means chapter X

Software Design Fundamentals	Brooks [3*]	Budgen [4*]	Gamma et al. [6*]	Sommerville [19*]	See also
Design Thinking	c1, c2, c3	c1, c2			[3, 18]
Context of Software Design		c13, c14		c19, c20	
Key Issues in Software Design					[2, 11]
Software Design Principles					[5, 9, 15, 18]
Software Design Processes		c3		c2, c7	[9]
High-level Design	c5	c6			[9]

Detailed Design				[9]
Software Design Qualities		c4		
Concurrency				c17
Control and Handling of Events				c21
Data Persistence				c6, c16
Distribution of Components				c17
Errors and Exception Handling, Fault Tolerance				c11
Integration and Interoperability		c11, c14, c16		
Assurance, Security and Safety				c10–c14
Recording Software Designs		c7, c8		[1]
Model-based Design		c7.3		c5.5
Structural Design Descriptions		c7, c10	c4	c5.3
Behavioral Design Descriptions		c9, c10	c5	c5.4
Design Patterns and Styles	c12	c15	c1, c2	c7.2 [6]
Specialized and Domain-specific languages				c15
Design Rationale	c16	c12		c6.1
Software Design Strategies				c3
General Strategies		c13		
Function-Oriented (“Structured”) Design		c9		
Data-Centered Design		c9		
Object-Oriented Design		c10		
User-Centered Design	c9			[14]
Component-Based Design (CBD)		c11, c16		c16
Event-Driven Design				[11, 13]
Aspect-Oriented Design				[11]

Constraint-Based Design	c11				
Other Methods				c18–c21	
Software Design Quality Analysis and Evaluation		c17		c24	
Design Reviews and Audits		c5.3			
Quality Attributes				c24	
Quality Analysis and Evaluation Techniques				c24	
Measures and Metrics		c5, c17		c24.5	
Verification, Validation and Certification				c7,c8	

FURTHER READINGS

Brooks, The Design of Design [3*]

Brooks, one of the pioneers of software engineering, provides a collection of essays and case studies on all aspects of software design.

REFERENCES

- [1] * Booch, G., J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, 2nd edition, Addison-Wesley, 2005.
- [2] Bosch, J., *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*, ACM Press, 2000.
- [3] * Brooks, F. *The Design of Design*, Addison-Wesley, 2010.
- [4] * Budgen, D. *Software Design: Creating Solutions for Ill-structured problems*, 3rd Edition CRC Press, 2021.
- [5] Dijkstra, E. W., On the role of scientific thought. 1974. <http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html>.
- [6] * Gamma, E. et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed, Addison-Wesley, 1994.
- [7] Gregor, S. and Jones, D., The anatomy of a design theory, Association for Information Systems, 2007.
- [8] IEEE Std 7000™-2021, IEEE Standard Model Process for Addressing Ethical Concerns during System Design.
- [9] ISO/IEC/IEEE 12207, *Systems and software engineering — Software life cycle processes*.
- [10] ISO/IEC/IEEE 24765, *Systems and software engineering — Vocabulary*.
- [11] Kiczales, G. et al., Aspect-Oriented Programming, *Proc. 11th European Conf. Object-Oriented Programming (ECOOP 97)*, Springer, 1997.
- [12] Luckham, D., *The power of events: an introduction to Complex Event Processing*, Addison-Wesley, 2002.
- [13] Mühl, G., Fiege, L. and Pietzuch, P., *Distributed event-based systems*, Springer-Verlag, 2006.
- [14] Nielsen, J. *Usability Engineering*, Morgan Kaufman, 1994.
- [15] Parnas, D.L. On the criteria to be used in decomposing systems into modules, *Communications of the ACM* **15**(12), 1053–1058, 1972.
- [16] Parnas, D.L. and Clements, P.C. A rational design process: How and why to fake it, *IEEE Transactions on Software Engineering* **12**(2), 251–257, 1986.
- [17] Parnas, D.L. and Weiss, D.M. Active design reviews: Principles and practices, *Journal of Systems & Software* **7**, 259–265, 1987
- [18] Ross, D.T., Goodenough, J.B., Irvine, A. Software Engineering: Process, Principles, and Goals, *IEEE Computer*, May 1975.
- [19] * Sommerville, I. *Software Engineering*, 10th edition, Pearson, 2016.

CHAPTER 4

SOFTWARE CONSTRUCTION

ABBREVIATIONS

API	Application programming interface
ASIC	Application-specific integrated circuit
BaaS	Backend as a service
CI	Continuous integration
COTS	Commercial off-the-shelf
CSS	Cascading style sheets
DSL	Domain-specific language
DSP	Digital signal processor
ESB	Enterprise service bus
FPGA	Field programmable gate array
GUI	Graphical user interface
HTML5	HyperText markup language version 5
IDE	Integrated development environment
IEEE	Institute of electrical and electronics engineers
ISO	International organization for standardization
J2EE	Java 2 platform, enterprise edition
KA	Knowledge area
MDA	Model-driven architecture
NPM	Node Package Manager
OMG	Object management group
PIM	Platform independent model
POSIX	Portable operating system interface
PSM	Platform-specific model
SDK	Software development kit
TDD	Test-driven development
UML	Unified Modeling Language
WYSIWYG	What you see is what you get

INTRODUCTION

Software construction refers to the detailed creation and maintenance of software through coding, verification, unit testing, integration testing and debugging.

The Software Construction Knowledge Area (KA) is linked to all the other KAs, but it is most strongly linked to the Software Design and Software Testing KAs because the software construction process involves significant design and testing. The process uses the design output and provides an input to testing (“design” and “testing” in this case referring to the activities, not the KAs). Boundaries among design, construction and testing (if any) vary depending on the software life cycle processes used in a project.

Although some detailed design might be performed before construction, much design work is performed during construction. Thus, the Software Construction KA is closely linked to the Software Design KA.

Also, throughout construction, software engineers both unit-test and integration-test their work. Thus, the Software Construction KA is closely linked to the Software Testing KA as well.

The Software Construction KA is also related to configuration management, quality, project management and computing, and thus to the relevant KAs.

First, software construction typically produces the highest number of configuration items that need to be managed in a software project (e.g., source files, documentation, test cases). Thus, the Software Construction KA is closely linked to the Software Configuration Management KA.

Second, while quality is important in all the KAs, code is a software project’s ultimate deliverable, and code is produced during construction. Thus, the Software Quality KA is closely linked to the Software Construction KA.

Third, while project management involve various software development tasks, software construction typically produces the most deliverables of a software project. Thus, the Software Construction KA is closely linked to the Software Engineering Management KA.

Fourth, since software construction requires knowledge of algorithms and coding practices, this KA is closely related to the Computing Foundations KA, which concerns the computer science foundations supporting software product design and construction.

BREAKDOWN OF TOPICS FOR SOFTWARE ARCHITECTURE

The breakdown of topics for the Software Architecture KA is shown in Figure 4-1.

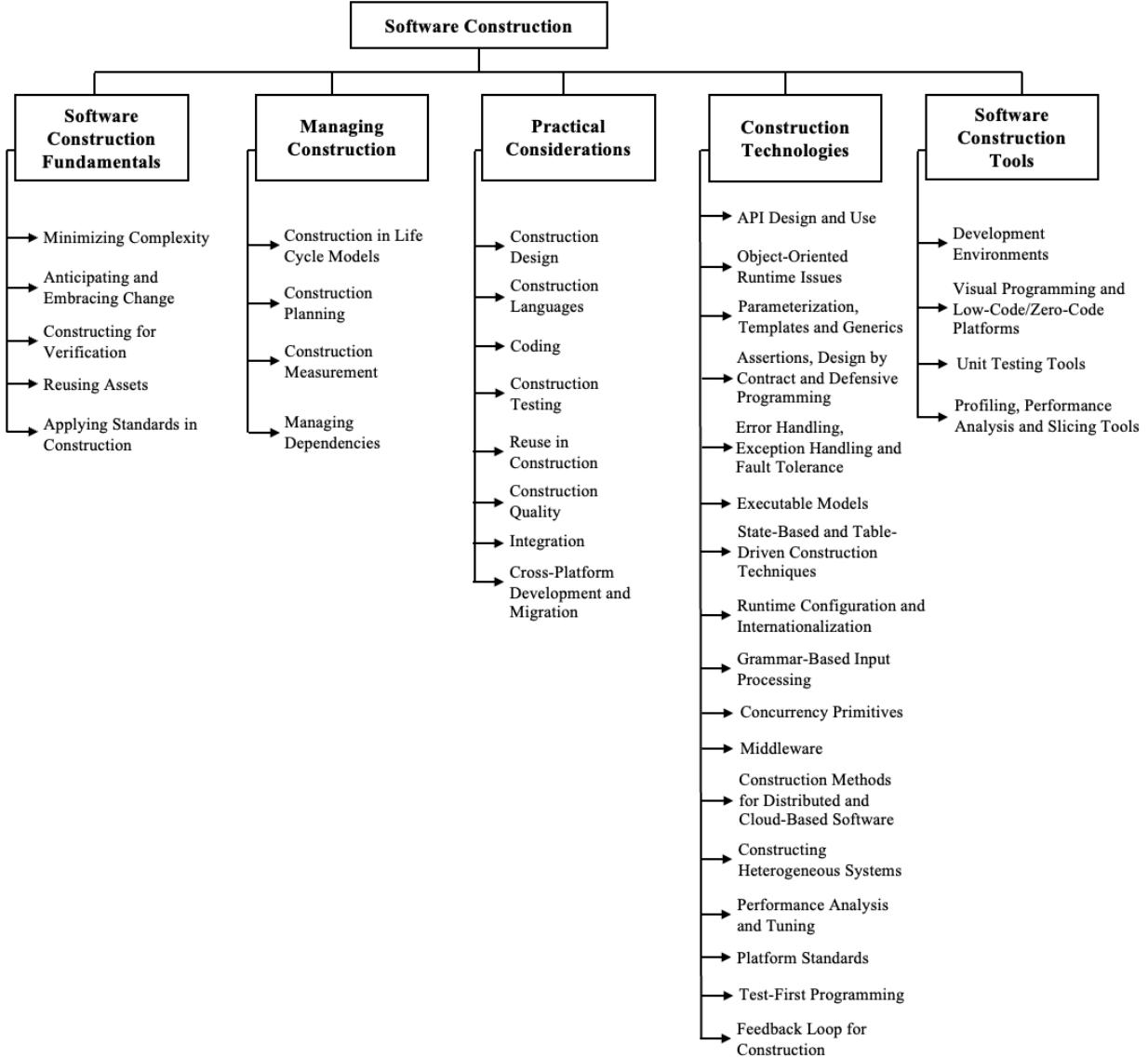


Fig. 1. Breakdown of Topics for the Software Construction KA

1. Software Construction Fundamentals

Software construction fundamentals include the following:

- Minimizing complexity
- Anticipating and embracing change
- Constructing for verification
- Reusing assets
- Applying standards in construction

The first four concepts apply to design as well as to construction. The following sections define these concepts and describe how they apply to construction.

1.1. Minimizing Complexity [1]

Most people have limited ability to hold complex structures and information in their working memories, especially over long periods. This greatly influences how people convey intent to computers and drives one of the key goals in software construction — to minimize complexity. The need to reduce complexity applies to essentially every aspect

of software construction and is particularly critical to testing software constructions.

Several types of complexity can affect software construction. Tools can be used to manage different aspects of the complexity of software components and their construction. For example, cyclomatic complexity is a static analysis measure of how difficult code is to test and understand. The tool, developed by Thomas J. McCabe, Sr., in 1976, calculates the number of linearly independent paths through a program's source code. Ideally, there should be at least that number of test cases. Other examples are tools like Make, which can build an application, or integrated development environments (IDEs) for entering, editing and compiling code. These tools help manage the complexity of the construction process.

In software construction, reduced complexity is achieved by creating simple and readable code rather than clever code. This is accomplished by using standards (see section 1.5, Standards in Construction), modular design (see section 3.1, Construction Design) and numerous other specific techniques (see section 3.3, Coding). Construction-focused quality techniques also support this (see section 3.6, Construction Quality).

1.2. Anticipating and Embracing Change [1, 2, 3]

Most software changes over time, and anticipating change drives many aspects of software construction; changes in the environments in which software operates also affect software in diverse ways. Anticipating change helps software engineers build extensible software, enhancing a software product without disrupting the underlying structure. Anticipating change is supported by many specific techniques (see section 3.3, Coding).

Moreover, today's business environments require many organizations to deliver and deploy software more frequently, faster and more reliably. Anticipating specific, necessary changes can be difficult, so software engineers should be careful to build flexibility and adaptability into the software to incorporate changes with less difficulty. These software teams should embrace change by adopting agile development, practicing DevOps, and by adopting continuous delivery and deployment practices. Such practices align the software development process and management with an evolutionary environment.

1.3. Constructing for Verification [1]

Constructing for verification builds software in such a way that faults can be readily found by the software engineers writing the software as well as by the testers and users during independent testing and operational activities. Specific techniques that support constructing for verification include following coding standards to support code reviews and unit testing, organizing code to support automated testing, and restricting the use of complex or difficult-to-understand language structures.

1.4. Reusing Assets [2]

Reuse means using existing assets to solve different problems. In software construction, typical assets that are reused include frameworks, libraries, modules, components, source code and commercial off-the-shelf (COTS) assets. Reuse has two closely related facets: *construction for reuse* and *construction with reuse*. The former means creating reusable software assets, whereas the latter means reusing software assets to construct a new solution. Reuse often transcends project boundaries, which means

reused assets can be constructed in other projects or organizations.

1.5. Applying Standards in Construction [1]

Applying external or internal development standards during construction helps achieve a project's efficiency, quality and cost objectives. Specifically, the choices of allowable programming language subsets and usage standards are important aids in achieving higher security.

Standards that directly affect construction issues include the following:

- Communication methods (e.g., standards for document formats and content)
- Programming languages (e.g., standards for languages like Java and C++)
- Coding standards (e.g., standards for naming conventions, layout and indentation)
- Platforms (e.g., interface standards for operating system calls)
- Tools (e.g., diagrammatic standards for notations like UML (Unified Modeling Language))

Use of external standards: Construction depends on external standards for construction languages, construction tools, technical interfaces and interactions between the Software Construction KA and other KAs. Standards come from numerous sources, including hardware and software interface specifications (e.g., object management group (OMG)) and international organizations (e.g., the institute of electrical and electronics engineers (IEEE) or the international organization for standardization (ISO)).

Use of internal standards: Standards may also be created on an organizational basis at the corporate level or for use on specific projects. These standards support coordinating group activities, minimizing complexity, anticipating change and constructing for verification.

2. Managing Construction

2.1. Construction in Life Cycle Models [1, 2, 3]

Numerous models have been created to develop software; some emphasize construction more than others.

Some models are more linear from the construction viewpoint, such as the waterfall and staged-delivery life cycle models. These models treat construction as an activity that occurs only after the completion of significant prerequisite work, including detailed requirements work, extensive design work and detailed planning. The more linear approaches emphasize the activities that precede construction (requirements and design) and create more distinct separations between activities. In these models, construction's main emphasis might be coding.

Other models, such as evolutionary prototyping and agile development, are more iterative. These approaches treat construction as an activity that occurs concurrently with or overlaps other software development activities (including requirements, design and planning). These approaches mix design, coding and testing activities, and they often treat the combination of activities as construction (see the Software Engineering Management and Software Process KAs).

The practices of continuous delivery and deployment further mix coding, testing,

delivery and deployment activities. In these practices, software updates made during construction activities are continuously delivered and deployed into the production environment. The whole process is fully automated by a deployment pipeline that consists of various testing and deployment activities.

Consequently, what is considered construction depends on the life cycle model used. In general, software construction is mostly coding and debugging, but it also involves construction planning, detailed design, unit testing, integration testing and other activities.

2.2. Construction Planning [1]

The choice of construction method is a key aspect of the *construction planning* activity. This choice affects the extent to which construction prerequisites are performed, the order in which they are performed and the degree to which they should be completed before construction work begins.

The approach to construction affects the project team's ability to reduce complexity, anticipate change and construct for verification. Each objective may also be addressed at the process, requirements and design levels, but the choice of construction method will influence them.

Construction planning also defines the order in which components are created and integrated, the integration strategy (for example, phased or incremental integration), the software quality management processes, the allocation of task assignments to specific software engineers, and other tasks, according to the chosen method.

2.3. Construction Measurement [1]

Numerous construction activities and artifacts can be measured, including code developed, modified, reused, and destroyed; code complexity; code inspection statistics; fault-fix and fault-find rates; effort; and scheduling. These measurements can be useful for managing construction, ensuring quality during construction and improving the construction process, among other uses (see the Software Engineering Process KA for more on measurement).

2.4. Managing Dependencies [2]

Software products often heavily rely on dependencies, including internal and external (commercial or open-source) dependencies, which allow developers to reuse common functionalities instead of reinventing the wheel and substantially improve developers' productivity. In addition, package managers (e.g., Maven in Java and NPM in JavaScript) are widely used to automate the process of installing, upgrading, configuring and removing dependencies.

The direct and indirect dependencies of software products constitute a dependency supply chain network. Any dependency in the supply chain network can introduce potential risk to software products and should be managed by developers or tools. Unnecessary dependencies should be avoided to improve build efficiency. License conflicts between dependencies and software products should be avoided to reduce legal risk. Propagation of dependencies' defects or vulnerabilities into software products should be avoided to improve the quality of software products. Regulations and monitoring mechanisms should be developed to prevent developers from introducing untrusted external dependencies.

3. Practical Considerations

Construction is an activity in which the software engineer often has to deal with sometimes chaotic, changing and even conflicting real-world constraints. Because of real-world constraints, practical considerations drive construction more than some other KAs, and software engineering is perhaps most craft-like in the construction activities compared with other activities.

3.1. Construction Design [1, 2]

Some projects allocate considerable design activity to construction, whereas others allocate design to a phase explicitly focused on design. Regardless of the exact allocation, some detailed design work occurs at the construction level, and that design work is dictated by constraints imposed by the real-world problem the software addresses.

Just as construction workers building a physical structure must make small modifications for unanticipated gaps in the builder's plans, software construction workers must make small or large modifications to flesh out software design details during construction.

The details of the design activity at the construction level are essentially the same as described in the Software Design KA, but they are applied at a smaller scale to algorithms, data structures and interfaces.

3.2. Construction Languages [1]

Construction languages include all forms of communication by which a human can specify an executable solution to a problem. Consequently, construction languages and their implementations (e.g., compilers) can affect software quality attributes such as performance, reliability and portability. As a result, they can seriously contribute to security vulnerabilities.

The simplest construction language is a *configuration language*, in which software engineers choose from a limited set of predefined options to create new or custom software installations. The text-based configuration files used in both the Windows and Unix operating systems are examples of this, and some program generators' menu-style selection lists constitute another example of a configuration language.

Toolkit languages are used to build applications from elements in toolkits (integrated sets of application-specific reusable parts); they are more complex than configuration languages. Toolkit languages may be explicitly defined as application programming languages, or the applications might be implied by a toolkit's set of interfaces.

Scripting languages are commonly used application programming languages. In some scripting languages, scripts are called *batch files* or *macros*.

Programming languages are the most flexible construction languages. They also contain the least amount of information about specific application areas and development processes. Therefore, they require the most training and skill to use effectively. The choice of programming language can greatly affect the likelihood of vulnerabilities being introduced during coding (e.g., uncritical use of C and C++ is questionable from a security viewpoint).

Three general notations are used for programming languages:

- Linguistic (e.g., C/C++, Java)
- Formal (e.g., Event-B)
- Visual (e.g., MATLAB)

Linguistic notations are distinguished in particular by the use of textual strings to represent complex software constructions. The combination of textual strings in patterns may have a sentence-like syntax. Properly used, each string should have a strong semantic connotation providing an immediate intuitive understanding of what happens when the software construction is executed.

Formal notations rely less on intuitive, everyday meanings of words and text strings and more on definitions backed by precise, unambiguous and formal (or mathematical) definitions. Formal construction notations and methods are at the semantic base of most system programming notations, where accuracy, time behavior and testability are more important than ease of mapping into natural language. Formal constructions also use precisely defined ways of combining symbols that avoid the ambiguity of many natural language constructions.

Visual notations rely much less on the textual notations of linguistic and formal construction and more on direct visual interpretation and placement of visual entities that represent the underlying software. Visual construction is somewhat limited by the difficulty of making “complex” statements using only the arrangement of icons on a display. However, these icons can be powerful tools in cases where the primary programming task is to build and “adjust” a visual interface to a program, the detailed behavior of which has an underlying definition.

Nowadays, domain-specific languages (DSLs) are widely used to build domain-specific applications. Unlike a general-purpose programming language, such as

C/C++ or Java, a DSL is designed for the application construction of a particular domain. Therefore, a DSL usually can be defined based on a higher level of abstraction of the target domain and can be optimized for a specific class of problems. Furthermore, A DSL usually can be expressed by visual notations defined by domain-specific concepts and rules.

3.3. Coding [1]

The following considerations apply to the software construction *coding* activity:

- Techniques for creating understandable source code, including naming conventions and source code layout
- Use of classes, enumerated types, variables, named constants and other similar entities
- Use of control structures
- Handling of error conditions — both anticipated and exceptional (e.g., input of bad data)
- Prevention of code-level security breaches (e.g., buffer overflows or array index bounds)
- Resource use through use of exclusion mechanisms and discipline in accessing serially reusable resources, including threads and database locks
- Source code organization into statements, routines, classes, packages or other structures
- Code documentation
- Code tuning

3.4. Construction Testing [1, 2]

Construction involves two forms of testing, which are often performed by the software engineer who wrote the code: unit testing and integration testing.

Construction testing aims to reduce the gap between when faults are inserted into the code and when those faults are detected, thereby reducing the cost incurred to fix them. In some instances, test cases are written after the code has been written. In other instances, test cases might be created before code is written.

Construction testing typically involves a subset of the various types of testing, described in the Software Testing KA. For instance, construction testing does not typically include system testing, alpha testing, beta testing, stress testing, configuration testing, usability testing, or other more specialized testing.

Two standards have been published on construction testing: IEEE Standard 829-1998, “IEEE Standard for Software Test Documentation,” and IEEE Standard 1008-1987, “IEEE Standard for Software Unit Testing.”

See sections 2.1.1 and 2.1.2 in the Software Testing KA for more specialized reference material.

3.5. Reuse in Construction [2]

Reuse in construction includes both construction for reuse and construction with reuse.

Construction for reuse creates software with the potential to be reused in the future for the present project or for other projects with a broad-based, multisystem perspective. Construction for reuse is usually based on variability analysis and design. To avoid the problem of code clones, developers should encapsulate reusable code fragments into well-structured libraries or components.

The tasks related to software construction for reuse during coding and testing are as follows:

- Variability implementation with mechanisms such as parameterization, conditional compilation and design patterns
- Variability encapsulation to make the software assets easy to configure and customize
- Testing the variability provided by the reusable software assets
- Description and publication of reusable software assets

Construction with reuse means creating new software by reusing existing software assets. The most popular reuse method is to reuse code from the libraries provided by the language, platform, tools or an organizational repository. Aside from these, many applications developed today use open-source libraries. In addition, reused and off-the-shelf software often have the same (or better) quality requirements as newly developed software (e.g., security level requirements).

The tasks related to software construction with reuse during coding and testing are as follows:

- Selecting reusable units, databases, test procedures or test data
- Evaluating code or test reusability
- Integrating reusable software assets into the current software
- Reporting reuse information on new code, test procedures or test data

The forms of reusable software assets are not limited to software artifacts that must be locally integrated. Nowadays, cloud services that provide various services through online interfaces such as RESTful application

programming interfaces (APIs) are widely used in applications. In the new cloud service model BaaS (backend as a service), applications delegate their backend implementations to cloud service providers — for example, utilities such as authentication, messaging and storage are usually provided by cloud providers.

Reuse is best practiced systematically, according to a well-defined, repeatable process. Systematic reuse can enable significant software productivity, quality and cost improvements. Systematic reuse is supported by methodologies such as software product line engineering and various software frameworks and platforms. Widely used frameworks such as Spring provide reusable infrastructures for enterprise applications so software teams can focus on application-specific business logic. Commercial platforms provide various reusable frameworks, libraries, components and tools to support application development to build their ecosystems.

3.6. Construction Quality [1, 2]

In addition to faults occurring during requirements and design activities, faults introduced during construction can cause serious quality problems (e.g., security vulnerabilities). These include not only faults in security functionality but also faults elsewhere that allow bypassing of the security functionality or create other security weaknesses or violations.

Numerous techniques exist to ensure the quality of code as it is constructed. The primary techniques used to ensure construction quality are the following:

- Unit testing and integration testing (see section 3.4, Construction Testing)

- Test-first development (see section 6.1.2 in the Software Testing KA)
- Use of assertions and defensive programming
- Debugging
- Inspections
- Technical reviews, including security-oriented reviews (see section 2.3 in the Software Quality KA)
- Static analysis (see section 2.2.1 of the Software Quality KA)

The specific technique or techniques selected depend on the software constructed and on the skill set of the software engineers performing the construction activities. Programmers should know good practices and common vulnerabilities (e.g., from widely recognized lists about common vulnerabilities). Automated static code analysis for security weaknesses is available for several common programming languages and can be used in security-critical projects.

Construction quality activities are differentiated from other quality activities by their focus. These activities focus on artifacts that are closely related to code — such as detailed design — as opposed to other artifacts that are less directly connected to the code, such as requirements, high-level designs and plans.

3.7. Integration [1, 2, 3]

During construction, a key activity is integrating individually constructed routines, classes, components and subsystems into a single system. In addition, a particular software system may need to be integrated with other software or hardware systems.

Concerns related to construction integration include planning the sequence in which components are integrated, identifying what

hardware is needed, creating scaffolding to support interim versions of the software, determining the degree of testing and quality work performed on components before they are integrated, and determining points in the project at which interim versions of the software are tested.

Programs can be integrated by means of either the phased or the incremental approach. Phased integration, also called *big bang* integration, entails delaying the integration of component software parts until all parts intended for release in a version are complete. Incremental integration is thought to offer many advantages over the traditional phased integration (e.g., easier error location, improved progress monitoring, earlier product delivery and improved customer relations). In incremental integration, the developers write and test a program in small pieces and then combine the pieces one at a time. Additional test infrastructure, including, for example, stubs, drivers and mock objects, is usually needed to enable incremental integration. In addition, by building and integrating one unit at a time (e.g., a class or component), the construction process can provide early feedback to developers and customers. Other advantages of incremental integration include easier error location, improved progress monitoring and more fully tested units, among others.

Nowadays, continuous integration (CI) has been widely adopted in practice. A software team using CI integrates its work frequently, leading to multiple integrations per day. CI is usually automated by a pipeline that builds and tests each integration to detect errors and provide fast feedback.

3.8. Cross-Platform Development and Migration [4]

Some applications, such as mobile applications, heavily rely on specific platforms (e.g., Apple, Android), which usually include operating systems, development frameworks and APIs. To support multiple platforms, the developers need to develop and build an application separately for each target platform using the corresponding program language and software development kit (SDK). However, multi-platform development in this way requires more time and cost and might cause different user experiences between different implementations.

Cross-platform development allows the developers to develop an application using a universal language and export it to various platforms. This usually can be done in two ways for mobile applications. One way is to generate native applications using specific tools that can compile the universal language into platform-specific formats. The other is to develop hybrid applications that combine web applications developed using languages like hypertext markup language version 5 (HTML5) and cascading style sheets (CSS) and native containers or wrappers for various operations systems.

For applications that are not developed in this way, developers may consider migrating the applications from one platform to another. The migration usually involves translation of different programming languages and platform-specific APIs and can be partially automated by tools.

4. Construction Technologies

4.1. API Design and Use [5]

An *API* is a set of signatures that are exported and available to the users of a library or a framework to write their applications. Besides signatures, an API should always include statements about the program's effects and/or behaviors (i.e., its semantics).

API design should make the API easy to learn and memorize, lead to readable code, be difficult to misuse, be easy to extend, be complete, and maintain backward compatibility. As the APIs usually outlast their implementations for a widely used library or framework, an API should be straightforward and stable, to facilitate client application development and maintenance.

API use involves selecting, learning, testing, integrating and possibly extending APIs provided by a library or framework (see section 3.5, Reuse in Construction).

4.2. Object-Oriented Runtime Issues [1]

Object-oriented languages support runtime mechanisms, including polymorphism and reflection. These runtime mechanisms increase the flexibility and adaptability of object-oriented programs.

Polymorphism is a language's ability to support general operations without knowing until runtime what kind of concrete objects the software will include. Because the program does not know the types of the objects in advance, the exact behavior is determined at runtime (called *dynamic binding*).

Reflection is a program's ability to observe and modify its structure and behavior at runtime. For example, reflection allows inspection of classes, interfaces, fields and methods at runtime without knowing their names at compile time. It also allows

instantiation of new objects at runtime and invocation of methods using parameterized class and method names.

4.3. Parameterization, Templates and Generics [6]

Parameterized types, also known as *generics* (Ada, Eiffel) and *templates* (C++), enable a type or class definition without specifying all the other types used. The unspecified types are supplied as parameters at the point of use. Parameterized types provide a third way (besides class inheritance and object composition) to compose behaviors in object-oriented software.

4.4. Assertions, Design by Contract and Defensive Programming [1]

An *assertion* is an executable predicate placed in a program — usually a routine or macro — that allows runtime checks of the program. Assertions are especially useful in high-reliability programs. They enable programmers to more quickly flush out mismatched interface assumptions, errors that creep in when code is modified, and other problems. Assertions are typically compiled into the code at development time and are later compiled out of the code so they don't degrade the performance.

Design by contract is a development approach in which preconditions and postconditions are included for each routine. When preconditions and postconditions are used, each routine or class is said to form a contract with the rest of the program. A contract precisely specifies the semantics of a routine and thus helps clarify its behavior. Design by contract is thought to improve the quality of software construction.

Defensive programming means to protect a routine from being broken by invalid inputs.

Common ways to handle invalid inputs include checking the values of all the input parameters and deciding how to handle bad inputs. Assertions are often used in defensive programming to check input values.

4.5. Error Handling, Exception Handling and Fault Tolerance [1]

How *errors* are handled affects software's ability to meet requirements related to correctness, robustness and other nonfunctional attributes. Assertions are sometimes used to check for errors. Other error-handling techniques — such as returning a neutral value, substituting the next piece of valid data, logging a warning message, returning an error code or shutting down the software — are also used.

Exceptions are used to detect and process errors or exceptional events. The basic structure of an exception is as follows: A routine uses *throw* to throw a detected exception, and an exception-handling block will *catch* the exception in a *try-catch* block. The try-catch block may process the erroneous condition or return control to the calling routine. Exception-handling policies should be carefully designed following common principles, such as including in the exception message all information that led to the exception, avoiding empty catch blocks, knowing the exceptions the library code throws, perhaps building a centralized exception reporter, and standardizing the program's use of exceptions.

Fault tolerance is a collection of techniques that increase software reliability by detecting errors and then recovering from them or containing their effects if recovery is not possible. The most common fault tolerance strategies include backing up and retrying, using auxiliary code and voting algorithms,

and replacing an erroneous value with a phony value that will have a benign effect.

4.6. Executable Models [7]

Executable models abstract away the details of specific programming languages and decisions about the software's organization. Different from traditional software models, a specification built in an executable modeling language like xUML (executable UML) can be deployed in various software environments without change. Furthermore, an executable-model compiler (transformer) can turn an executable model into an implementation using a set of decisions about the target hardware and software environment. Thus, constructing executable models is a way of constructing executable software.

Executable models are one foundation supporting the model-driven architecture (MDA) initiative of the OMG. An executable model is a way to specify a platform-independent model (PIM); a PIM is a model of a solution to a problem that does not rely on any implementation technologies. Then a platform-specific model (PSM), which is a model that contains the details of the implementation, can be produced by weaving together the PIM and the platform on which it relies.

4.7. State-Based and Table-Driven Construction Techniques [1]

State-based programming, or *automata-based programming*, is a programming technology that uses finite-state machines to describe program behaviors. A state machine's transition graphs are used in all stages of software development (specification, implementation, debugging and documentation). The main idea is to construct computer programs in the same way technological processes are automated.

State-based programming is usually combined with object-oriented programming, forming a new composite approach called *state-based, object-oriented programming*.

A *table-driven* method is a schema that uses tables to display information rather than convey information with logic statements (such as *if* and *case*). When used in appropriate circumstances, table-driven code is simpler than complicated logic and easier to modify. When using table-driven methods, the programmer addresses two issues: what information to store in the table or tables and how to efficiently access information in the table.

4.8. Runtime Configuration and Internationalization [1]

To achieve more flexibility, a program is often constructed to support its variables' late binding time. For example, *runtime configuration* binds variable values and program settings when the program is running, usually by updating and reading configuration files in a just-in-time mode.

Internationalization is the technical activity of preparing a program, usually interactive software, to support multiple locales. The corresponding activity, *localization*, modifies a program to support a specific local language. Interactive software may contain dozens or hundreds of prompts, status displays, help messages, error messages and so on. The design and construction processes should accommodate string and character set issues, including which character set is used, what kinds of strings are used, how to maintain the strings without changing the code and how to translate the strings into different languages with minimal impact on the processing code and the user interface.

4.9. Grammar-Based Input Processing [1, 8]

Grammar-based input processing involves syntax analysis, or *parsing*, of the input token stream. It creates a data structure (called a *parse tree* or *syntax tree*) representing the input data. The inorder traversal of the parse tree usually gives the expression just parsed. Next, the parser checks the symbol table for programmer-defined variables that populate the tree. After building the parse tree, the program uses it as an input to the computational processes.

4.10. Concurrency Primitives [9]

A *synchronization primitive* is a programming abstraction provided by a programming language or the operating system that facilitates concurrency and synchronization. Well-known concurrency primitives include semaphores, monitors and mutexes.

A *semaphore* is a protected variable or abstract data type that provides a simple but useful abstraction for controlling access to a common resource by multiple processes or threads in a concurrent programming environment.

A *monitor* is an abstract data type that presents a set of programmer-defined operations executed with mutual exclusion. A monitor contains the declaration of shared variables and procedures or functions that operate on those variables. The monitor construct ensures that only one process at a time is active in the monitor.

A *mutex* (mutual exclusion) is a synchronization primitive that grants exclusive access to a shared resource by only one process or thread at a time.

4.11. Middleware [5, 8]

Middleware is a broad classification for software that provides services above the operating system layer yet below the application program layer. Middleware can provide runtime containers for software components to provide message passing, persistence and a transparent location across a network. Middleware can be viewed as a connector between the components using the middleware. Modern message-oriented middleware usually provides an enterprise service bus (ESB) that supports service-oriented interaction and communication among multiple software applications.

4.12. Construction Methods for Distributed and Cloud-Based Software [2, 9]

A *distributed system* is a collection of physically separate, possibly heterogeneous computer systems networked to provide the users with access to the resources the system maintains. The construction of distributed software is distinguished from traditional software construction by issues such as parallelism, communication and fault tolerance.

Distributed programming typically falls into several basic architectural categories: client-server, three-tier architecture, n-tier architecture, distributed objects, loose coupling or tight coupling (see section 5.6 in the Computing Foundations KA and section 2.2 in the Software Architecture KA).

Nowadays, more applications are migrated to the cloud. *Cloud-based software* often adopts microservice architecture and container-based deployment. In addition to traditional distributed software issues, cloud-based software developers also need to consider cloud infrastructure issues such as use of an

API gateway, service registration and discovery.

4.13. Constructing Heterogeneous Systems [8]

Heterogeneous systems consist of various specialized computational units of different types, such as digital signal processors (DSPs), microcontrollers and peripheral processors. These computational units are independently controlled and communicate with one another. Embedded systems are typically heterogeneous systems.

The design of heterogeneous systems may require combining several specification languages to design different system parts (hardware/software codesign). The key issues include multilanguage validation, co-simulation and interfacing.

During the hardware/software codesign, software and virtual hardware development proceed concurrently through stepwise decomposition. The hardware part is usually simulated in field programmable gate arrays (FPGAs) or application-specific integrated circuits (ASICs). The software part is translated into a low-level programming language.

4.14. Performance Analysis and Tuning [1]

Code efficiency — determined by architecture, detailed design decisions, and data structure and algorithm selection — influences execution speed and size. *Performance analysis* investigates a program's behavior using information gathered as the program executes to identify possible hot spots in the program to be improved.

Code *tuning*, which improves performance at the code level, modifies correct code to make

it run more efficiently. Code tuning usually involves only small changes that affect a single class, a single routine or, more commonly, a few lines of code. A rich set of code tuning techniques is available, including those for tuning logic expressions, loops, data transformations, expressions and routines. Using a low-level language is another common technique for improving hot spots in a program.

4.15. Platform Standards [4, 8, 9]

Platform standards enable programmers to develop portable applications that can be executed in compatible environments without changes. Platform standards usually involve standard services and APIs that compatible platform implementations must use. Typical examples of platform standards are Java 2 platform, enterprise edition (J2EE); the portable operating system interface (POSIX) standard for operating systems, which represents a set of standards implemented primarily for Unix-based operating systems; and HTML5, which defines the standards for developing web applications that can run on different environments (e.g., Apple iOS, Android).

4.16. Test-First Programming [1, 2]

Test-first programming (also known as test-driven development (TDD)) is a popular development style in which test cases are written before any code. Test-first programming can usually detect defects earlier and correct them more easily than traditional programming styles. Furthermore, writing test cases first forces programmers to think about requirements and design before coding, thus exposing requirements and design problems sooner.

4.17. Feedback Loop for Construction [3]

Early and continuous feedback for the construction activity is one of the most

important advantages of agile development and DevOps. Agile development provides early feedback for construction through frequent iterations in the development process. DevOps provides even faster feedback from the operation, allowing the developers to learn how well their code performs in production environments. This fast feedback is achieved through techniques and practices in the DevOps pipeline, such as automated building and testing, canary release, and A/B testing.

5. Software Construction Tools

5.1. Development Environments [1]

A *development environment*, or *integrated development environment* (IDE), provides comprehensive facilities to programmers for software construction by integrating a set of development tools. The programmers' choice of development environment can affect software construction efficiency and quality.

Besides basic code editing functions, modern IDEs often offer other features, like compilation and error detection within the editor, integration with source code control, build/test/debugging tools, compressed or outline views of programs, automated code transforms, and support for refactoring.

Nowadays, cloud-based development environments are available in public or private cloud services. These environments can provide all the features of modern IDEs and even more (e.g., containerized building and deployment), powered by the cloud.

5.2. Visual Programming and Low-Code/Zero-Code Platforms [1]

Visual programming allows users to create programs by manipulating visual program elements graphically. As a visual programming tool, a GUI (graphical user

interface) builder enables the developer to create and maintain GUIs in a WYSIWYG (what you see is what you get) mode. A GUI builder usually includes a visual editor that enables the developer to design forms and windows and manage the layout of the widgets with drag, drop and parameter setting features. Some GUI builders can automatically generate the source code corresponding to the visual GUI design. Because GUI applications usually follow the event-driven style (in which events and event handling determine the program flow), GUI builder tools usually provide code generation assistants, which automate the most repetitive tasks required for event handling. The supporting code connects widgets with the outgoing and incoming events that trigger the functions providing the application logic. Some modern IDEs provide integrated GUI builders or GUI builder plug-ins. There are also many stand-alone GUI builders.

Visual programming and other rapid application development tools have evolved into *low-code/zero-code platforms*. These platforms allow developers to build complete applications visually through a drag-and-drop interface and with minimal hand-coding. They are usually based on the principles of model-driven design, visual programming and code generation. The difference between low-code development and zero-code development lies in hand-coding; the former requires a little hand-coding, whereas the latter requires practically none.

5.3. Unit Testing Tools [1, 2]

Unit testing verifies the functioning of software modules in isolation from other

separately testable software elements (for example, classes, routines, components). Unit testing is often automated. Developers can use unit testing tools and frameworks to extend and create an automated testing environment. For example, the developer can code criteria into the test with unit testing tools and frameworks to verify the unit's correctness under various data sets. Each test is implemented as an object, and a test runner runs the tests. Failed test cases will be automatically flagged and reported during the test execution.

5.4. Profiling, Performance Analysis and Slicing Tools [1]

Performance analysis tools are usually used to support code tuning. The most common performance analysis tools are *profiling tools*. An execution profiling tool monitors the code while it runs and records how often each statement is executed or how much time the program spends on each statement or execution path. Profiling the code while it runs gives insight into how the program works, where the hot spots are and where the developers should focus the code tuning efforts.

Program slicing involves computing the set of program statements (i.e., the program slice) that might affect the values of specified variables at some point of interest, which is called a slicing criterion. Program slicing can be used for locating error sources, program understanding and optimization analysis. Program slicing tools compute program slices for various programming languages using static or dynamic analysis methods.

MATRIX: TOPICS VS. REFERENCE MATERIAL

	McConnell, 2004 [1]	Sommerville , 2016 [2]	Kim et al., 2021 [3]	Heitkötter et al., 2013 [4]	Clem ents et al., 2010 [5]	Gam ma et al., 1994 [6]	Mellor and Balcer, 2002 [7]	Null and Lobur , 2006 [8]	Silber schatz et al., 2008 [9]
1. Software Construction Fundamentals									
1.1. Minimizing Complexity	c2, c3, c7-c9, c24, c27, c28, c31, c32, c34								
1.2. Anticipating and Embracing Change	c3-c5, c24, c31, c32, c34	c1, c3, c9	c1						
1.3. Constructing for Verification	c8, c20-c23, c31, c34								
1.4. Reuse		c15							
1.5. Standards in Construction	c4								
2. Managing Construction									
2.1. Construction in Life Cycle Models	c2, c3, c27, c29	c3, c7	c1						
2.2. Construction Planning	c3, c4, c21, c27-c29								
2.3. Construction Measurement	c25, c28								
2.4. Managing Dependencies		c25							
3. Practical Considerations									
3.1. Construction Design	c3, c5, c24	c7							
3.2. Construction Languages	c4								
3.3. Coding	c5-c19, c25-c26								
3.4. Construction Testing	c22, c23	c8							
3.5. Reuse in Construction		c15, c16							
3.6. Construction Quality	c8, c20-c25	c8, c24							
3.7. Integration	c29	c8	c11						
3.8. Cross-Platform Development and Migration				c					
4. Construction Technologies									
4.1. API Design and Use					c7				
4.2. Object-Oriented Runtime Issues	c6, c7								
4.3. Parameterization, Templates and Generics						c1			
4.4. Assertions, Design by Contract and Defensive Programming	c8, c9								
4.5. Error Handling, Exception Handling and Fault Tolerance	c3, c8								
4.6. Executable Models							c1		
4.7. State-Based and Table-Driven Construction Techniques	c18								
4.8. Runtime Configuration and Internationalization	c3, c10								
4.9. Grammar-Based Input Processing	c5							c8	
4.10. Concurrency Primitives									c6
4.11. Middleware					c1			c8	

4.12. Construction Methods for Distributed and Cloud-Based Software		c17, c18							c2
4.13. Constructing Heterogeneous Systems								c9	
4.14. Performance Analysis and Tuning	c25, c26								
4.15. Platform Standards				c				c10	c1
4.16. Test-First Programming	c22	c8							
4.17. Feedback Loop for Construction			c3, c16						
5. Software Construction Tools									
5.1. Development Environments	c30								
5.2. Visual Programming and Low-Code/Zero-Code Platforms	c30								
5.3. Unit Testing Tools	c22	c8							
5.4. Profiling, Performance Analysis and Slicing Tools	c25, c26								

REFERENCES

- [1] McConnell, S., *Code Complete*, 2nd edition, Redmond, WA: Microsoft Press, 2004.
- [2] Sommerville, I., *Software Engineering*, 10th edition, Addison-Wesley, 2016.
- [3] Kim, G., et al., *The DevOps Handbook: How to Create World-Class Agility, Reliability & Security in Technology Organizations*, 2nd edition, IT Revolution, 2021.
- [4] Heitkötter, H., Hanschke, S., Majchrzak, T.A., Evaluating Cross-Platform Development Approaches for Mobile Applications, 2013, in Cordeiro, J., Krempels, K.H. (eds.), *Web Information Systems and Technologies*. WEBIST 2012. Lecture Notes in Business Information Processing, vol. 140, Springer, Berlin, Heidelberg.
- [5] Clements, P., et al., *Documenting Software Architectures: Views and Beyond*, 2nd edition, Boston: Pearson Education, 2010.
- [6] Gamma, E., et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st edition,
- [7] Reading, MA: Addison-Wesley Professional, 1994.
- [8] Mellor, S.J., and Balcer, M.J., *Executable UML: A Foundation for Model-Driven Architecture*, 1st edition, Boston: Addison-Wesley, 2002.
- [9] Null, L., and Lobur, J., *The Essentials of Computer Organization and Architecture*, 2nd edition, Sudbury, MA: Jones and Bartlett Publishers, 2006.
- Silberschatz, A., et al., *Operating System Concepts*, 8th edition, Hoboken, NJ: Wiley, 2008.

CHAPTER 5

SOFTWARE TESTING

ACRONYMS

AI	Artificial Intelligence
API	Application Program Interface
ARINC	Aeronautical Radio Incorporated
ATDD	Acceptance Test-Driven Development
CMMI	Capability Maturity Model Integration
CSS	Cascading Style Sheets
DICOM	Digital Imaging and Communications in Medicine
DL	Deep Learning
DU	Definition and Use
EBSE	Evidence-Based Software Engineering
ECS	Ecosystem
ETSI	European Telecommunications Standards Institute
FHIR	Fast Healthcare Interoperability Resources
GDPR	General Data Protection Regulation
GPS	Global Positioning System
GUI	Graphical User Interface
HIL	Hardware-In-the-Loop

HIPAA	Health Insurance Portability and Accountability Act
HL7	Health Level Seven
IoT	Internet of Things
KPI	Key Performance Indicator
MC/DC	Modified Condition Decision Coverage
ML	Machine Learning
MTTR	Mean Time to Recovery
OAT	Orthogonal Array Testing
ODC	Orthogonal Defect Classification
RUP	Rational Unified Process
SoS	System of Systems
SPI	Software Process Improvement
SPICE	Software Process Improvement and Capability Determination
SUT	System Under Test
TDD	Test-Driven Development
TMMi	Test Maturity Model integration
UI	User Interface

INTRODUCTION

Software testing consists of the *dynamic* validation that a *System Under Test (SUT)* provides *expected* behaviors on a *finite* set of *test cases* suitably *selected* from the usually infinite execution domain.

In the above statement, italicized words correspond to key issues in the Software Testing Knowledge Area (KA). Those terms are discussed below.

- *System Under Test:* This term can refer to the testing object, which could be a program, a software product, an application, a service-oriented application (e.g., web services, microservices), middleware (HW/SW), a

- services composition, a system, a System of Systems (SoS), or an Ecosystem (ECS).
- *Test Case*: A test case is the specification of all the entities that are essential for the execution, such as input values, execution and timing conditions, testing procedure, and the expected outcomes (e.g., produced values, state changes, output messages). Input values alone are not always sufficient to specify the test cases because the SUT might react to the same input with different behaviors, depending, for instance, on the SUT state or environmental conditions. A set of test cases is usually called a *test suite*.
 - *Dynamic*: Dynamic validation requires executing the SUT on a test suite. Static techniques complement dynamic testing, and they are covered in the Software Quality KA.¹
 - *Finite*: Even in a simple SUT, executing all the possible test cases (i.e., exhaustive testing) could require months or years. Consequently, in practice, testing targets a subset of all possible test cases determined by different criteria. Testing always implies a trade-off between limited resources and schedules on the one hand and inherently unlimited test requirements on the other.
 - *Selected*: Identifying the most suitable selection criteria under given conditions is a complex problem. Different techniques can be considered and combined to tackle that problem, such as risk analysis, software requirements, cost reduction, quality attributes satisfaction, prioritization, and fault detection. The many proposed test techniques differ in how the test suite is selected, and software engineers must be aware that different selection criteria might yield vastly different degrees of effectiveness.
 - *Expected*: For each executed test case, it must be possible, although it might not be easy, to decide whether the observed SUT outcomes match the expected ones. Indeed, the observed behavior may be checked against user needs (commonly referred to as testing for validation), against a specification (testing for verification), or, perhaps, against the foreseen behavior from implicit requirements or expectations. (See Section 4.3, Acceptance Criteria-Based Requirements Specification, in the Software Requirements KA.)

As reflected in this discussion, software testing is a pervasive and holistic activity involving all the steps

of any process development life cycle (e.g., traditional or left-shift development). The remainder of this chapter presents the basics of software testing and its challenges, issues, and commonly accepted practices and solutions.

BREAKDOWN OF TOPICS FOR SOFTWARE TESTING

Figure 1 shows the breakdown of topics for the Software Testing KA. The Matrix of Topics vs. Reference Material provides a more detailed breakdown at the end of this KA. The first topic, Software Testing Fundamentals, covers the basic definitions in software testing, the basic terminology and key issues, and software testing's relationship with other activities.

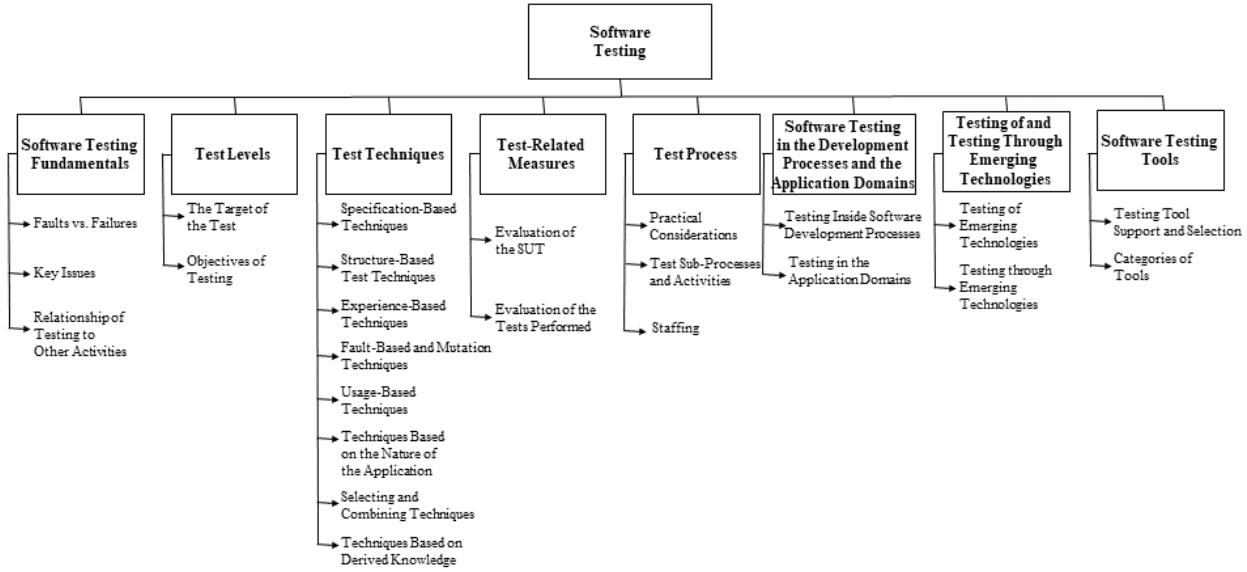
The second topic, Test Levels, contains two (orthogonal) subtopics. The first subtopic, The Target of the Test, lists the levels into which the testing of large software is traditionally subdivided, and the second subtopic, Objectives of Testing, discusses testing for specific conditions or properties. Not all types of testing apply to every software product, nor has every possible type been listed. The Target of the Test and Objectives of Testing together determine how the test suite is identified, both regarding its consistency (How much testing is enough for achieving the stated objective?) and its composition (Which test cases should be selected for achieving the stated objective?). (However, usually, “for achieving the stated objective” remains implicit, and only the first part of the two questions above is posed.) Criteria for addressing the first question are *test adequacy criteria*, whereas those used for addressing the second question are the *test selection criteria*.

Several Test Techniques have been developed in the past few decades, and new ones are still being proposed. Therefore, the third topic covers generally accepted and standardized techniques.

Test-Related Measures are dealt with in the fourth topic, while the issues relative to the Test Process are covered in the fifth.

Software Testing in the Development Processes and the Application Domains is described in the sixth topic, and Testing of and Testing Through Emerging Technologies are described in the seventh topic.

¹ It is worth noting that terminology is not uniform among different communities, and some use the term *testing* to refer to static techniques as well.



Finally, Software Testing Tools are presented in topic eight.

1. Software Testing Fundamentals

[1*, c1, c2; 2*, c8; 14*, c7]

This section provides an overview of the main testing issues and the relationship of testing to the other activities. Most of the testing terms used here are also defined. A more comprehensive overview of the testing and testing-related terminology can be found in the cited references.

1.1 Faults vs. Failures

[1*, c1s5; 2*, c1; 14*, c1s3]

Many terms are used in the software engineering literature to describe a malfunction: notably *fault* (see, for comparison, *defect* in Section 3.2, Defect Characterization, in the Software Quality KA), *failure* and *error*. It is essential to distinguish between the *cause* of a malfunction (for which the term *fault* is used here) and an undesired effect observed in the system's delivered service (a *failure*). Indeed, there might well be faults in the software that never manifest as failures. (See Theoretical and Practical Limitations of Testing in Section 1.2.8.) Thus, testing can reveal failures, but the faults causing them are what can and must be removed. However, a failure's cause cannot always be unequivocally identified. No theoretical criteria exist to definitively determine, in general, the fault that caused an observed failure. The fault might have to be modified to remove the failure, but other modifications might also work. To avoid ambiguity, we could refer to *failure-causing inputs* instead of

faults — those sets of inputs that cause a failure to appear.

1.2 Key Issues

This subsection provides an overview of the main testing issues.

1.2.1. Test Case Creation

[1*, c12s1, c12s3, 2*, c8]

Test case creation or *generation* creates the test suite useful for testing the SUT for specific purposes (e.g., adequacy, accuracy, or assessment). Because test case generation is among the most important and intensive software testing activities, it is usually supported by approaches, techniques, and tools to automate the process.

1.2.2. Test Selection and Adequacy Criteria

[1*, c1s14, c6s6, c12s7, 2*, c8]

A test selection criterion is a means of selecting test cases or determining that a test suite is sufficient for a specified purpose. Test case selection aims to reduce the cardinality of the test suites while keeping the same effectiveness in terms of coverage or fault detection rate. Test adequacy criteria can be used to decide when sufficient testing is accomplished.

1.2.3 Prioritization/Minimization

[4, part 2, part 3, c5]

Suitable strategies for test case selection or prioritization can be adopted to improve testing efficacy. Test case *prioritization* aims to define a test execution order according to some criteria (e.g., coverage, fault detection rate, similarity, and risk), so

those tests with a higher priority are executed before those with a lower priority. Test case *minimization* usually aims to reduce a test suite by removing redundant test cases according to some criterion or purpose.

1.2.4. Purpose of Testing

[1*, c13s11, c11s4, 2*, c8]

Different well-defined purposes can guide testing activity; it is only by considering a specific purpose that a test suite can be generated (selected), executed, and evaluated (see Section 2 for more details).

1.2.5. Assessment and Certification

[4, part 1, c5; 2*, c7, c25; 8]

Testing needs to focus on specific (mandatory) prescriptions, such as requirements, laws, and standards. Test cases should be generated and executed to provide evidence useful for evaluating and/or certifying adherence to the selected prescriptions. Usually, *assessment* and *certification* of the test results include verifying that the test cases have been derived and generated using baseline requirements, adopting a configuration control process, and using repeatable processes.

1.2.6. Testing for Quality Assurance/Improvement

[1*, c16s2; 4, part 1, c5; 9]

Testing has many aspects, including quality improvement and assurance. These characteristics involve planned and systematic supporting processes and activities leveraging confidence that the SUT fulfills established technical or quality requirements. Thus, quality *improvement* and *assurance* involve defining methods, tools, skills, and practices to achieve the specific quality level and objectives. The list of the main quality characteristics that testing can measure or assess is reported in ISO/IEC 25010:2011 [9]. (See also Section 1.3.2, Software Product Quality, in the Software Quality KA.)

1.2.7. The Oracle Problem

[1*, c1s9, c9s7]

An important testing component is the oracle. Indeed, a test is meaningful only if it is possible to decide its observed outcome. An *oracle* can be any human or mechanical agent that decides whether the SUT behaved correctly in each test and according to the expected outcomes. Consequently, the oracle provides a “pass” or “fail” verdict. The oracle cannot always decide; in these cases, the test output is classified as inconclusive. There are many kinds of oracles — for example, unambiguous requirements specifications,

behavioral models, and code annotations. Automation of mechanized oracles can be difficult and expensive.

1.2.8. Theoretical and Practical Limitations

[1*, c2s7]

Testing theory warns against ascribing unjustified confidence to a series of successful tests. Unfortunately, most established results of the testing theory are negative results in that they state what is not achieved as opposed to what is achieved. The most famous quotation on this point is the Dijkstra aphorism that “program testing can be used to show the presence of bugs, but never to show their absence” [3]. The obvious reason for this is that complete testing is not feasible in realistic software.

1.2.9. The Problem of Infeasible Paths

[1*, c4s7]

Infeasible paths are control flow paths that cannot be exercised by any input data (i.e., test cases). Managing the infeasible paths can help reduce the time and resources devoted to testing. They are a significant problem in path-based testing, particularly in the automated derivation of test cases to exercise control flow paths. Additionally, infeasible paths can also be connected to the analysis and detection process for security vulnerabilities and can improve accuracy.

1.2.10. Testability

[1*, c17s2]

The term *software testability* has two related but different meanings. On the one hand, it refers to the ease with which a given test coverage criterion can be satisfied; on the other hand, it is defined as the likelihood, possibly measured statistically, that a test suite will expose a failure if the software is faulty. Both meanings are important.

1.2.11 Test Execution and Automation

[4, part 1, c4]

An important challenge of testing is to improve attainable automation, either by developing advanced techniques for generating the test inputs or, beyond test generation, by finding innovative support procedures to (fully) automate the different testing activities — for instance, to increase the number of test cases generated or executed.

1.2.12. Scalability

[1*, c8s7]

Scalability is the software’s ability to increase and scale up on its nonfunctional requirements, such as

load, number of transactions, and volume of data. Scalability is also connected to the complexity of the platform and environment in which the program runs, such as distributed, wireless networks and virtualized environments, large-scale clusters, and mobile clouds.

1.2.13 Test Effectiveness

[1* c1s1; 2* c8s1; 8]

Evaluating the SUT, measuring a testing technique's efficacy, and judging whether testing can be stopped are important issues for software testing, and they may require defining and selecting the proper test effectiveness measures.

1.2.14 Controllability, Replication, and Generalization

[1* c12s12; 4, part 2, c7]

Specific aspects of testing include the following:

- *Controllability* refers to the transition of testing activities from the laboratory (i.e., controlled conditions) to reality (i.e., uncontrolled conditions).
- *Replication* refers to the ability for different people to perform the same testing activities. The purpose is to verify whether a given testing theory works, at least in the laboratory.
- The *generalization* of testing is connected to external validity — i.e., the extent to which the test approach can be applied to broader settings or target populations. The generalizability of the software testing can be important for managing the testing activities (in terms of cost and effort) and increasing confidence in the test results.

1.2.15 Off-Line vs. Online Testing

[10, c3]

The testing process can be executed in two settings: *off-line* and *online*. Usually, the SUT is validated in an environment without external interaction in off-line testing, whereas the SUT interacts with the real application environment in online testing. The test cases are either manually or automatically derived in both cases, and the expected outcomes are used to assess the SUT.

1.3. Relationship of Testing to Other Activities

- Software testing is related to but different from static software quality management techniques, proofs of correctness, debugging, and program construction. However, it is informative to consider testing from the viewpoint of software quality analysts and certifiers. For further discussion, see the following:

- Testing vs. Static Software Quality Management Techniques: See Section 2.2.1, Static Analysis Techniques, in the Software Quality KA.
- Testing vs. Quality Improvement/Accurance: See Section 1.3.2, Software Product Quality, in the Software Quality KA.
- Testing vs. Correctness Proofs and Formal Verification: See the Software Engineering Models and Methods KA.
- Testing vs. Debugging: See Construction Testing in the Software Construction KA and Debugging Tools and Techniques in the Computing Foundations KA.
- Testing vs. Program Construction: See Construction Testing in the Software Construction KA.
- Testing vs. Security: See the new KA: Software Security.
- Testing vs. Effort Estimation: See the Software Engineering Management KA.
- Testing vs. Legal Issues: See the Software Engineering Professional Practice KA.

2. Test Levels

[1*, c1s13; 2*, c8s1]

Software testing is usually performed at different *levels* throughout development and maintenance. Levels can be distinguished based on the object of testing, the *target*, or on the purpose or *objective* (of the test level).

2.1. The Target of the Test

[1*, c1s13, 2*, c8s1]

The target of the test can vary depending on the SUT, the conditions of the environment, and the budget/time devoted to the testing activity. Four test stages can be distinguished: unit, integration, system, and acceptance. These four test stages do not imply any development process, nor is any one of them assumed to be more important than the other three.

2.1.1. Unit Testing

[1*, c3, 2*, c8]

Unit testing verifies the functioning in isolation of SUT elements that are separately testable. Depending on the context, these could be the individual subprograms or components, a subsystem, or a composition of SUT components. Typically, but not always, the person who wrote the code conducts the unit testing.

2.1.2. Integration Testing

[1*, c7, 2*, c8]

Integration testing verifies the interactions among SUT elements (for instance, components, modules, or subsystems). Integration strategies involve the incremental (and systematic) integration of the SUT elements considering either identified functional threads or architecture specifications. Typical integration testing strategies are top-down, bottom-up, mixed (or sandwiched), and the big bang. They focus on different perspectives of the level at which SUT elements are integrated. Integration testing is a continuous activity that can be performed at each development stage. It may target different aspects, such as interoperability (e.g., compatibility or configuration) of the SUT elements or with the external environment. External interfaces to other applications, utilities, hardware devices or operating environments can also be considered.

2.1.3. System Testing

[1*, c8, 2*, c8]

System testing concerns testing the behavior of the SUT (according to the definition of Section 1). Effective unit and integration testing should have identified many SUT defects. In addition, system testing is usually considered appropriate for assessing non-functional system requirements, such as security, privacy, speed, accuracy, and reliability. (See Functional and Non-Functional Requirements in the Software Requirements KA and Software Quality Requirements in the Software Quality KA.)

2.1.4. Acceptance Testing

[1*, c1s7, 2*, c8s4]

Acceptance testing targets the deployment of a SUT. Its main goal is to verify that the SUT satisfies the requirements and the end-users expectations. Generally, it is run by or with the end-users to perform those functions and tasks for which the software was built. For example, this testing activity could target usability testing or operational acceptance. Defining acceptance tests before implementing the corresponding functionality is a key activity of the Acceptance Test-Driven Development (ATDD). (See the Software Requirements KA, Section 4.3.)

2.2. Objectives of Testing

[1*, c1s7]

Testing is conducted considering specific *objectives*, which are stated (more or less) explicitly and with varying degrees of precision. Stating the testing objectives in precise, quantitative terms supports measurement and control of the test process.

Testing can be aimed at verifying different properties. For example, test cases can be designed to check that

the functional specifications are correctly implemented, which is variously referred to in the literature as conformance testing, correctness testing or functional testing. However, several other non-functional properties may be tested as well, including performance, reliability, and usability. (See Models and Quality Characteristics in the Software Quality KA.)

Other important testing objectives include but are not limited to reliability measurements, identification of security and privacy vulnerabilities, and usability evaluation; different approaches would be necessary depending on the objective. Note that, in general, the test objectives vary with the test target; different purposes are addressed at different levels of testing.

The subtopics listed below are those most cited in the literature.

2.2.1. Conformance Testing

[1*, c10s4]

Conformance testing aims to verify that the SUT conforms to standards, rules, specifications, requirements, design, processes, or practices.

2.2.2 Compliance Testing

[1*, c12s3]

Compliance testing aims to demonstrate the SUT's adherence to a law or regulation. Usually, compliance testing is forced by an external regulatory body.

2.2.3. Installation Testing

[1*, c12s2]

Often, after system and acceptance testing is completed, and the SUT has been installed in the target environment, the SUT is verified. *Installation testing* can be viewed as system testing conducted in the operational environment of hardware configurations and other operational constraints. Installation procedures may also be verified.

2.2.4. Alpha and Beta Testing

[1*, c13s7, c16s6, 2*, c8s4]

Before the SUT is released, it is sometimes given to a small, selected group of potential users for trial use (*alpha testing*) and/or to a larger set of representative users (*beta testing*). These users report problems with the product. Alpha testing and beta testing are often uncontrolled and are not always referred to in a test plan.

2.2.5. Regression Testing

[1*, c8s11, c13s3; 4, part 1, c5]

According to the definition reported in [5], *regression testing* is the “selective retesting of a SUT to verify that modifications have not caused unintended effects and that the SUT still complies with its specified requirements.” In practice, the approach is designed to show that the SUT still passes previously passed tests in a test suite (in fact, it is sometimes referred to as *non-regression testing*). In some cases, a trade-off must be made between the assurance given by regression testing every time a change is made and the resources required to perform the regression tests. This can be quite time-consuming because of the many tests that might be executed. Regression testing can be conducted at each test level described in Section 2.1. It may involve functional and non-functional testing, such as reliability, accessibility, usability, maintainability, conversion, migration, and compatibility testing.

Regression testing may involve selection (see Section 1.2.2) and minimization (see Section 1.2.3) of test cases, as well as the adoption of prioritization approaches (see Section 2.2.6) to existing test suites.

Regression testing is a fundamental activity of Agile, DevOps, Test-Driven Development (TDD), and Continuous Development. It is usually performed after integration testing and before deployment to production or operation.

2.2.6. Prioritization Testing

[1*, c12s7]

Test case prioritization aims to schedule test cases to increase the rate of fault detection, the likelihood of revealing faults, the coverage of code under test, and the SUT’s reliability. Typically, prioritization testing relies on heuristics, and its performance might vary according to the SUT, the environment, and the available test cases. Among the different prioritization proposals, similarity-based prioritization is one of the most commonly adopted. In this approach to prioritization, test cases are prioritized starting from those most dissimilar according to a predefined distance function.

2.2.7. Non-functional Testing

[2*, c8]

Non-Functional testing targets the validation of non-functional aspects (such as performance, usability, or reliability), and it is performed at all test levels. At the state of the practice, there are hundreds of non-functional testing techniques that include but are not limited to the following:

- Performance Testing [4, part 1]: Performance testing verifies that the software meets the specified performance requirements and assesses performance characteristics (e.g., capacity and response time).
- Load Testing [4, part 1]: Load testing focuses on validating the SUT’s behavior under load pressure conditions to discover problems (e.g., deadlocks, racing, buffer overflows and memory leaks) or reliability, stability, or robustness violations. It aims to assess the rate at which different service requests are submitted to the SUT.
- Stress Testing [1*, c8s8]: Stress testing aims to push the SUT beyond its capabilities by generating a load greater than what the system is expected to handle.
- Volume Testing [4, part 1]: Volume testing targets the assessment of the SUT’s internal storage limitations and its ability to exchange data and information.
- Failover Testing [1*, c17s2; 2*, c8]: Failover testing validates the SUT’s ability to manage heavy loads or unexpected failure to continue typical operations (e.g., by allocating extra resources). Failover testing is also connected with recoverability validation.
- Reliability Testing [1*, c15; 2*, c11]: Reliability testing evaluates the SUT’s reliability by identifying and correcting faults. Reliability testing observes the SUT in operation or exercises the SUT by using test cases according to statistical models (operational profiles) of the different users’ behaviors. Usually, reliability is assessed through reliability growth models. The continuous development processes (such as DevOps) are currently facilitating the adoption of reliability testing in the various iterations for improving final SUT quality.
- Compatibility Testing [4, part 1; 10, c3]: Compatibility testing is used to verify whether the software can collaborate with different hardware and software facilities or with different versions or releases.
- Scalability Testing [1*, c8s7; 2* c17]: Scalability testing evaluates the capability to use and learn the system and the user documentation. It also focuses on the system’s effectiveness in supporting user tasks and the ability to recover from user errors. This testing is particularly important in distributed or high-performance systems.
- Elasticity Testing [2* c17]: Elasticity testing assesses the ability of the SUT (such as cloud and distributed systems) to rapidly expand or shrink

compute, memory, and storage resources without compromising the capacity to meet peak utilization. Some elasticity testing objectives are to control behaviors, to identify the resources to be (un)allocated, to coordinate events in parallel, and to evaluate scalability.

- Infrastructure Testing [8, annex H]: Infrastructure testing tests and validates infrastructure components to reduce the chances of downtime and improve the performance of the IT infrastructure.
- Back-to-Back Testing [5]: IEEE/ISO/IEC Standard 24765 defines back-to-back testing as “testing in which two or more variants of a program are executed with the same inputs, the outputs are compared, and errors are analyzed in case of discrepancies.”
- Recovery Testing [1*, c14s2]: Recovery testing is aimed at verifying software restart capabilities after a system crash or other disasters.

2.2.8. Security Testing

[2*, c13; 4, part 4, annex A]

Security testing focuses on validating that the SUT is protected from external attacks. More precisely, it verifies the confidentiality, integrity, and availability of the systems and their data. Usually, security testing includes validation against misuse and abuse of the software or system (negative testing). (See Security Testing in the Software Security KA.)

2.2.9. Privacy Testing

[2*, c13, c14]

Privacy testing is devoted to assessing the security and privacy of users' personal data to prevent local attacks. It specifically assesses privacy and information-sharing policies, as well as the validation of decentralized management of users' social profiles and data storage solutions. (See Legal Issue in the Software Engineering Professional Practice KA.)

2.2.10. Interface and Application Program Interface (API) Testing

[2*, c8s1; 14*, c7s12; 4, part 5, c4, c7]

Interface defects are common in complex systems. *Interface testing* aims to verify whether the components' interface provides the correct exchange of data and control information. Usually, the test cases are generated from the interface specification. A specific interface testing objective is to simulate the use of APIs by end-user applications. That involves generating parameters of the API calls, setting conditions of the external environment, and defining internal data that affect the API.

2.2.11. Configuration Testing

[1*, c8s5]

Where the SUT is built to serve different users, *configuration testing* verifies the software under specified configurations.

2.2.12. Usability and Human-Computer Interaction Testing

[2* c8s4; 19*, c6; 4, part 4, annex A]

The main task of *usability and human-computer interaction testing* is to evaluate how easy it is for end-users to learn to use the software. It might involve testing the software functions that support user tasks, the documentation that aids users, and the system's ability to recover from user errors. (See User-Centered Design in the Software Design KA.)

3. Test Techniques

[1*, c1s15; 4, part 4]

Over the years, different testing techniques have been developed to increase the SUT's overall quality [4, part 4]. These techniques attempt to propose systematic procedures and approaches for generating or selecting the most suitable test suites for detecting as many failures as possible.

Testing techniques can be classified by considering different key aspects such as specification, structure, and experience [4, part 4]. Additional classification sources can be the faults to be discovered, the predicted use, the models, the nature of the application, or the derived knowledge. For instance, model-based testing [7; 4, part 1] refers to all the testing techniques that use the concept of a model representing behavioral specification, the SUT's structure, or the available knowledge and experience. However, classification overlapping is possible, and one category might deal with combining two or more techniques.

Alternative classifications that rely on the degree of information about the SUT are available in the literature. Indeed, in the specification-based techniques, also known as *black-box* techniques, the generation of test cases is based only on the SUT's input/output behavior, whereas in the structure-based, also called *white-box* (or *glass-box* or *clear-box*), techniques, the test cases are generated using the information about how the SUT has been designed or coded.

As some testing techniques are used more than others, the remainder of the section presents the standard

testing techniques and those commonly adopted at the state of the practice.

3.1. Specification-Based Techniques

[1*, c6s2; 4, part 4]

The underlying idea of *specification-based techniques* (sometimes also called domain testing techniques) is to select a few test cases from the input domain that can detect specific categories of faults (also called domain errors). These techniques check whether the SUT can manage inputs within a certain range and return the required output.

3.1.1. Equivalence Partitioning

[1*, c9s4]

Equivalence partitioning involves partitioning the input domain into a collection of subsets (or equivalence classes) based on a specified criterion or relation. This criterion or relation may be different computational results, a relation based on control flow or data flow, or a distinction made between valid inputs that are accepted and processed by the SUT and invalid inputs, such as out-of-range values, that are not accepted and should generate an error message or initiate error processing. A representative test suite (sometimes containing only one test case) is usually taken from each equivalence class.

3.1.2. Boundary-Value Analysis

[1*, c9s5; 4, part 4]

Test cases are chosen on or near the boundaries of the input domain of variables, with the underlying rationale that many faults tend to concentrate near the extreme values of inputs. An extension of this technique is *robustness testing*, wherein test cases are also chosen outside the input domain of variables to test program robustness in processing unexpected or erroneous inputs.

3.1.3. Syntax Testing

[1*, c10s11, 2*, c5; 4, part 4]

The Syntax Testing techniques, also known as formal specification-based techniques, rely on the SUT specifications in a formal language. (See Formal Methods in the Software Engineering Models and Methods KA.) This representation permits automatic derivation of functional test cases and, at the same time, provides an oracle for checking test results.

3.1.4. Combinatorial Test Techniques

[1*, c9s3; 4, part 4]

The Combinatorial Test Techniques systematically derive the test cases that cover specific parameters of

values or conditions. According to [4, part 4], the commonly used combinatorial test techniques are All-Combinations Testing, Pair-Wise Testing, Each Choice Testing, and Base Choice Testing. All-combinations testing focuses on all the possible input combinations, whereas its subset, also called *t-wise* testing, considers every possible combination of *t* input. In this case, more than one pair is derived (i.e., by including higher-level combinations). Pair-wise testing is a specific combinatorial testing technique where test cases are derived by combining values of every pair of an input set. These techniques are also known as Orthogonal Array Testing (OAT).

3.1.5. Decision Table

[1*, c9s6; 1*, c13s6; 4, part 4]

Decision tables (or trees) represent logical relationships between conditions (roughly, inputs) and actions (roughly, outputs). Usually, they are widely adopted for knowledge representation (e.g., Machine Learning (ML)). Test cases are systematically derived by considering every possible combination of conditions and their corresponding resultant actions. A related technique is *cause-effect graphing*. Currently, shift-left development processes are taking advantage of this kind of testing technique because these techniques are useful for documenting the test results and factors that can affect them.

3.1.6. Cause-Effect Graphing

[1*, c1s6; 4, part 3, part 4]

Cause-effect graphing techniques rely on logical networks that map a set of causes to a set of effects by systematically exploring the possible combinations of input conditions. They identify the effects and link the effects to their causes through model graphs. Cause-effect graphing techniques are used in testing because they allow specification analysis, the identification of the relevant input conditions or causes, the consequent transformations, and the output conditions.

3.1.7. State Transition Testing

[1*, c10; 4, part 4]

Techniques based on Finite-State Machines (*State Transition Testing* techniques in [4, part 4]) focus on representing the SUT with a finite-state machine. In this case, the test suite is derived to cover the states and transitions according to a specific coverage level.

3.1.8. Scenario-Based Testing

[2*, c8s3, c19s3; 4, part 4; 7]

A model in this context is an abstract (formal) representation of the SUT or its software requirements. (See Modeling in the Software

Engineering Models and Methods KA.) *Scenario-based testing* is used to validate requirements, check their consistency, and generate test cases focused on the SUT's behavioral aspects. (See Types of Models in the Software Engineering Models and Methods KA.) The key components of scenario-based testing are the notation used to represent the model of the software or its requirements, workflow models or similar models, the test strategy or algorithm used for test case generation, the supporting infrastructure for the test execution, and the evaluation of test results compared to expected results. Because of the complexity of the techniques, scenario-based testing approaches are often used with test automation harnesses.

Among scenario-based testing, workflow models can also be used to graphically represent the sequence of activities performed by humans and/or software applications. In this case, each sequence of actions constitutes one workflow (also called a scenario). Usually, it is important to ensure that both typical and alternate workflows are also tested. For example, business process testing is part of this scenario-based technique. In this case, the special focus is on the roles in a workflow specification.

3.1.9. Random Testing

[1*, c9s7; 4, part 4]

In this approach, test cases are generated purely at random. This testing falls under the heading of input domain testing because the input domain must be known to be able to pick random points within it. *Random testing* provides a relatively simple approach to test automation. Enhanced forms of random testing (such as adaptive random testing) have been proposed in which other input selection criteria direct the random input sampling.

Currently, under the name of fuzzy testing, the random selection of invalid and unexpected inputs and data is extensively used in cybersecurity to find hackable software bugs, coding errors, and security loopholes. (See also Sections 2.2.8 and 8.2.)

3.1.10. Evidence-Based

[10, c6s2]

Evidence-based software engineering (EBSE), which follows a rigorous research approach, is the *best* solution for a practical problem. EBSE includes the following phases:

- Identifying the evidence and forming a question
- Tracking down the best evidence to answer the question

- Critically analyzing the evidence in light of the problem that the evidence should help solve.

EBSE principles can also be applied to the testing process. For that purpose, the widely used approaches that allow identifying and aggregating evidence are systematic mapping studies and systematic reviews.

3.1.11. Forcing Exception

[5]

Test cases are specifically conceived for checking whether the SUT can manage a predefined set of exceptions/errors, such as data exception, operation exception, overflow exception, protection exception or underflow exception. Testing techniques usually focus on negative test scenarios (i.e., test cases that are able to force the generation of error messages).

3.2. Structure-Based Test Techniques

[4, part 4]

Structure-Based Test Techniques (sometimes called *code-based test techniques*) focus on the code and its structure. Structure-Based Test Techniques can be performed at different levels (such as code development, code inspection, or unit testing) and can include static testing (such as code inspection, code walkthrough, and code review), dynamic testing (like statement coverage, branch coverage, and path coverage), or code complexity measurement (e.g., using techniques like cyclomatic complexity [12]).

3.2.1. Control Flow Testing

[1*, c4; 4, part 4]

Control flow testing covers all the statements, branches, decisions, branch conditions, modified condition decision coverage (MC/DC), blocks of statements, or specific combinations of statements in a SUT. The strongest of the control flow-based criteria is path testing, which aims to execute all entry-to-exit control flow paths in a SUT's control flow graph. Because exhaustive path testing is generally not feasible because of loops, other less stringent criteria focus on coverage of paths that limit loop iterations, such as statement coverage, branch coverage, and condition/decision testing. The adequacy of such tests is measured in percentages; for example, when all branches have been executed at least once by the tests, 100% branch coverage has been achieved.

3.2.2. Data Flow Testing

[1*, c5; 4, part 4]

In *data flow testing*, the control flow graph is annotated with information about how the variables are defined, used, and killed (undefined). Commonly adopted data flow testing techniques are All-Definitions Testing, All-C-Uses Testing, All-P-Uses

Testing, All-Uses Testing and All-DU-Paths Testing. The strongest data flow testing criterion is the All-DU-Paths Testing, where all definition and use (DU) paths need to be covered. This is because it requires executing, for each variable, every control flow path segment from a definition of that variable to the use of that definition. However, weaker strategies such as all-definitions and all-uses are used to reduce the number of paths required.

3.2.3. Reference Models for Structure-Based Test Techniques

[1*, c4]

Although not a technique, a SUT's control structure can be graphically represented using a flow graph to visualize *structure-based test techniques*. A flow graph is a directed graph, the nodes and arcs of which correspond to program elements. (See Graphs and Trees in the Mathematical Foundations KA.) For instance, nodes may represent statements or uninterrupted sequences of statements, and arcs may represent the transfer of control between nodes.

3.3. Experience-Based Techniques

[4, part 1, part 4]

The generation of the most suitable test suite may depend on different factors, such as human knowledge of the SUT and its context and his/her experience and intuition. In the following section, the commonly adopted experience-based techniques are briefly introduced.

3.3.1. Error Guessing

[1*, c9s8; 4, part 4]

In *error guessing*, software engineers design test cases specifically to anticipate the most plausible faults in each SUT. Good sources of information are the history of faults discovered in earlier projects and the software engineer's expertise.

3.3.2. Exploratory Testing

[4, part 1]

Exploratory testing is defined as simultaneous learning, test design and test execution. The test cases are not defined in advance but are dynamically designed, executed, and modified according to the collected evidence and test results, such as observed product behavior, peculiarities of the SUT, the domain and the environment, the failure process, the types of possible faults and failures, and the risk associated with a particular product. Usually, the intuition, knowledge, and expertise of the personnel in charge of performing the exploratory testing can affect the testing effectiveness. Exploratory testing is widely

used in left-shift development (such as Agile). (See Section 5.4.2.)

3.3.3. Further Experience-Based Techniques

[4, part 4; 13]

At the state of the practice, *experience-based techniques* may include further approaches such as Ad Hoc-based, knowledge-based and ML-based testing techniques.

Ad Hoc testing is a widely used technique in which test cases are derived by relying on the software engineer's skill, intuition, and experience with similar programs. It can be useful for identifying test cases that are not easily generated by more formalized techniques. Typical Ad Hoc methodologies are the following:

- Monkey testing runs randomly generated test cases to cause the program to stop.
- Buddy testing generates test cases by using internal architecture knowledge and testing specific knowledge.
- Pair testing involves two individuals. One generates and runs the test cases; the other observes and analyzes the testing process. Pair testing allows generating test cases with broad and better test coverage.
- Gamification aims to convert testing tasks to components of gameplay. By applying specific techniques (such as engaging practitioners or crowdsourcing complex testing tasks), gamification can substantially improve software testing practice and, consequently, SUT quality.
- Quick testing, in which a very small test suite is selected and executed, guarantees that no failure can be experienced because of SUT components that are not fully operational.
- Smoke testing (also known as Build Verification Testing) ensures that the SUT's core functionalities behave properly. It also guarantees that the SUT is operational before the planned testing begins. In addition, smoke testing prevents failures because of the test environment (e.g., because artifacts or packages are not properly built). Smoke testing is also considered a special case of quick testing.

Knowledge-based testing and ML-based testing exploit (formal or informal) knowledge about the SUT or derive it from observations of SUT executions for defining its behavioral models (such as ontologies or decision tables) (see Section 3.6.1), rules, and non-functional properties. In addition, Knowledge-based testing and ML-based testing specify the testing needs

and identify test objectives for which test cases are generated.

3.4. Fault-Based and Mutation Techniques

[1*, c1s14, 1* c3s5; 5]

With different degrees of formalization, *fault-based* testing techniques devise test cases specifically to reveal likely or predefined fault categories. A *fault model* can be introduced that classifies the different faults to better focus the test case generation or selection. In this context, a variety of platforms and development processes (e.g., waterfall, spiral and Agile) consider the Orthogonal Defect Classification (ODC) a valid methodology for collecting semantic information about the different defects and reducing the time and effort of the root cause analysis.

Mutation Testing was originally conceived as a technique to evaluate test suites (see Section 4.2, Evaluation of the Tests Performed) in which a mutant is a slightly modified version of the SUT (also called *gold*), differing from it by a small syntactic change. Every test case exercises both the gold version and all generated mutants. If a test case succeeds in identifying the difference between the gold version and a mutant, the latter is said to be “killed.” The underlying assumption of mutation testing, the coupling effect, is that more complex but real faults will be found by looking for simple syntactic faults. For the technique to be effective, many mutants must be automatically generated and executed systematically [6]. Mutation testing is also a testing criterion in itself. Test cases are randomly generated until enough mutants have been killed, or tests are specifically designed to kill surviving mutants. In the latter case, mutation testing can also be categorized as a structure-based technique. Mutation testing has been used effectively for generating fuzzy testing. A more recent application of the mutation process is metamorphic testing, a technique that has become increasingly popular in addressing some ML systems’ testing challenges. In this case, the modifications (called also morph) are applied to the inputs so a relationship can connect the previous input (and its output) to the new morphed input (and its output).

3.5. Usage-Based Techniques

[1*, c15s5]

Usage-based techniques usually rely on a usage model or profiles. In this case, the testing environment needs to represent the actual operational environment, and the sequence of test case execution should reproduce the SUT usage by the target stakeholder. Statistical sampling is used for simulating the execution of many test cases. Thus, sometimes, the term *random testing* is also associated with these techniques. Usage-based

statistical testing is applied more during the acceptance testing stage.

3.5.1. Operational Profile

[1*, c15s5, 2*, c11]

Testing based on *operational profiles* aims at generating test cases that might estimate the reliability of the SUT or part of it. Therefore, the goal is to infer from the observed test results the future reliability of the software (when it is in use). Because the established reliability strictly depends on the operating profile, the main difficulty (and cost) in using this testing approach comes from the operational profile derivation. Therefore, one possible solution is to assign to the input the probabilities or profiles according to their frequency of occurrence in actual operation.

3.5.2. User Observation Heuristics

[19*, c5, c7; 4, part 4, annex A]

Specialized heuristics, also called *usability inspection methods*, are applied to systematically observe system use under controlled conditions to determine how well people can use the system and its interfaces. Usability heuristics include cognitive walkthroughs, claims analysis, field observations, thinking aloud, and even indirect approaches such as user questionnaires and interviews.

3.6. Techniques Based on the Nature of the Application

[2* c16, c17, c18, c20, c21; 14*, c4s8; 8]

The above techniques apply to all kinds of software. Additional test derivation and execution techniques are based on the nature of the software being tested. Examples are the following:

- Object-oriented software
- Component-based software
- Web-based software
- Concurrent programs
- Protocol-based software
- Communication systems
- Real-time systems
- Safety-critical systems
- Service-oriented software
- Open-source software
- Embedded software
- Cloud-based software
- Blockchain-based software
- Big data-based software
- AI/ML/DL-based software
- Mobile apps
- Security and privacy-preserving software

In some cases, standards such as ISO/IEC/IEEE 29119 [4, part 4, part 5] provide examples and support for specifying test cases, automating their execution, and maintaining the test suites, such as the case of the Keyword-Driven Testing [4, part 5].

3.7. Selecting and Combining Techniques

[14*, c7s12; 10; 4, part 5]

Combining different testing techniques has always been a well-grounded means to assure the required level of SUT quality. Currently, especially in left-shift developments, methodologies for adaptive combinations of testing techniques are part of the state of the practice. The goal is to improve the effectiveness of testing processes by learning from experience and, at the same time, adapting the technique selection to the current testing session.

3.7.1. Combining Functional and Structural

[1*, c9; 4, part 5]

Scenario-based and structure-based test techniques are often contrasted as *functional* vs. *structural* testing. These two approaches to test case selection are nowadays seen as complements, as they use different sources of information and have been shown to highlight different problems. Depending on the different organizational constraints, such as budgetary considerations, they could be combined.

3.7.2. Deterministic vs. Random

[1*, c9s6]

Test cases can be selected in a deterministic way, according to many techniques, or randomly drawn from some distribution of inputs, such as is usually done in reliability testing. Several analytical and empirical comparisons have been conducted to analyze the conditions that make one approach more effective than the other.

3.8. Techniques Based on Derived Knowledge

[2*, c19, c20; 14*, c7]

Testing techniques can integrate evidence and knowledge from different research areas and contexts. For this, approaches and methodologies are used to support testing activity and improve its effectiveness. Currently, innovative approaches include using digital twins or simulation methodologies and frameworks, exploiting ML and gamification facilities, and using (simulated) neuronal networks.

4. Test-Related Measures

[2*, c24s5; 14*, c10; 4, part 4]

Testing techniques are like tools that help in achieving specific test objectives. To evaluate whether a test objective is reached, well-defined measures are

needed. Measurement is usually considered fundamental to quality analysis. Measurement may also be used to optimize test planning and execution. Test management can use several different process measures to monitor progress. (See Software Engineering Measurement in the Software Engineering Management KA for information on measurement programs. See Software Measurement in the Software Engineering Process KA for information on measures.)

According to the definition in [4, part 4], testing techniques can be classified according to the degree of coverage they can achieve. Coverage may vary from 0% to 100%, excluding possible infeasible tests (i.e., tests that cannot be executed). Thus, for each specification-based, structure-based, and experience-based test technique, the associated coverage measures and the procedure for evaluating that coverage must be determined. Examples of coverage measures could be the percentage of branches covered in the program flow graph or the percentage of functional requirements exercised among those listed in the specifications document.

It is important to consider that monitoring facilities can dynamically compute the ratio between covered elements, and the total number may also be considered. Additionally, especially in the case of structure-based testing techniques, appropriate instrumentation of the SUT may also be necessary.

However, the proposed set of testing measures can also be classified from different viewpoints — from the point of view of those providing and allowing an evaluation of the SUT based on the observed test outputs and of those that evaluate the thoroughness or effectiveness of the executed test suites.

4.1. Evaluation of the SUT

[2*, c24s5]

Usually, indicators (i.e., measurable information) can be used to determine whether a SUT is performing as expected and achieving its expected outcomes. The indicators, sometimes known as Key Performance Indicators (KPIs), are strongly connected with the adopted evaluation measures, methods, data analysis and reporting.

4.1.1. SUT Measurements That Aid in Planning and Designing Tests

[14*, c10; 10, c6; 4, part 1]

All the testing measures proposed in [4, part 4] can be used for planning and guiding testing activities. Additionally, in the shift-left development process,

specific measures, such as Deployment Frequency, Lead Time, Mean Time to Recovery (MTTR), and Change Failure Rate, are also commonly adopted to plan and manage the testing activities and results.

4.1.2. Fault Types, Classification and Statistics

[1* c13s4, c13s5, c13s6]

The testing literature is rich in classifications and taxonomies of faults that can be generic or specific to a context or quality attributes (such as the usability defect classification, the taxonomy of HW/SW security and privacy vulnerabilities and attacks, and the classification of cybersecurity risks). To make testing more effective, it is important to know which types of faults may be found in the SUT and the relative frequency with which these faults have occurred in the past. This information can be useful in making quality predictions and in process improvement (See Characterization in the Software Quality KA).

4.1.3. Fault Density

[1*, c13s4; 14*, c10s1]

Traditionally, a SUT can be evaluated by counting discovered faults as the ratio between the number of faults found and the SUT size. Because of the semantics-based definition of faults, additional measurements can be considered, such as fault depth (the minimal number of fault removals needed to make a SUT correct) and fault multiplicity (the number of atomic changes needed to repair a single fault).

4.1.4. Life Test, Reliability Evaluation

[1*, c15, 2*, c11; 14*, c1s3]

A statistical estimate of software reliability, which can be obtained by observing reliability achieved, can be used to evaluate a SUT and decide whether testing can be stopped or the SUT is mature enough to be a candidate for the next left-shift development release. Reliability evaluation is taking a pivotal role in the Cloud (and fog) contexts [18].

On the one hand, validation and verification proposals are focusing on maintaining the high level of reliability and availability required by the cloud (fog) services. On the other, testing activities are exploiting the computational power of the cloud (fog) environment to speed up the reliability evaluation and drastically reduce its costs.

4.1.5. Reliability Growth Models

[1*, c15, 2* c11s5]

Reliability growth models predict reliability based on observed failures. They assume, in general, that when the faults that caused the observed failures have been fixed (although some models also accept imperfect fixes), the product's reliability will increase. There are many published reliability growth models. Notably, these models are divided into *failure-count* and *time-between-failure* models.

4.2. Evaluation of the Tests Performed

[4, part 4, c6]

The behavior of SUT is generally verified by executing test suites, which are pivotal in finding defects. Therefore, from both the researchers' and practitioners' perspectives, a fundamental part of software testing is comparing test suites. Usually, evaluating the test suites means comparing techniques for test case generation that produce the test cases. Different criteria are used for that purpose, such as coverage criteria or mutation analysis criteria.

4.2.1. Fault Injection

[1*, c2s5]

In *fault injection*, some faults are artificially introduced into the SUT before testing. When a test suite is executed, some of these injected faults are revealed, as are, possibly, some faults that were already there. In theory, depending on which and how many artificial faults are discovered, the testing effectiveness can be evaluated, and the remaining number of genuine faults can be estimated. In practice, statisticians question the distribution and representativeness of injected faults relative to genuine faults and the small sample size on which any extrapolations are based. Some also argue that this technique should be used with great care because inserting faults into the SUT incurs the obvious risk of leaving them there.

4.2.2. Mutation Score

[1*, c3s5; 6]

In mutation testing, the test suite effectiveness measure is calculated as the ratio of killed mutants to the number of generated mutants. The higher the test suite effectiveness value, the better, since it indicates a stronger ability to discover the most real injected faults.

4.2.3. Comparison and Relative Effectiveness of Different Techniques

[1*, c1s7; 5; 9]

Relative effectiveness compares different testing techniques against a specific property, such as the number of tests needed to find the first failure, the ratio

of the number of faults found through testing to all the faults found during and after testing, and how much reliability was improved. Several studies have already been conducted to compare different techniques analytically and empirically according to each notion of property (or effectiveness) defined.

5. Test Process

[4, part 1, part 2, part 3; 2* c8]

Testing concepts, strategies, techniques and measures need to be integrated into a defined and controlled test planning process to test output evaluation. The test process supports testing and provides guidelines to those responsible for different testing activities to ensure the test objectives are met cost-effectively.

As described in [4, part 2], the test process is a multi-layered process activity that includes the test specification at the organizational, management and dynamic levels. The organizational test process defines the steps for creating and maintaining test specifications, such as organizational test policies, strategies, processes, procedures, and other assets [4, part 2].

The test management process defines the steps necessary for management: planning, monitoring and control, and completion.

Finally, the dynamic test process specifies the steps for design and implementation, environment setup and maintenance, execution, and test incident reporting.

In the remainder of this section, some practical considerations about the test process specification, management, and execution, as well as a summary of the test sub-processes and activities included in the organizational, management and dynamic levels as in [4, part 2], are provided.

5.1. Practical Considerations

[4, part 1]

Testing processes should allow the automation of different testing phases and should rely on the controllability, traceability, replicability, and risk/cost estimation of the performed activities. In the remainder of this section, commonly applied test steps are described, compatible with and applicable to all life cycle models. (See Software Life Cycles in the Software Engineering Process KA.)

5.1.1. Attitudes/Egoless Programming

[1*, c16; 2*, c3]

An important element of successful testing is a collaborative attitude toward testing and Quality Assurance (QA) activities. Managers have a key role in fostering a favorable reception toward failure discovery and correction during software development and maintenance. For instance, in shift-left development processes, such as Agile, communication and collaboration among testers and developers are considered vital for achieving successful testing results.

5.1.2. Test Guides and Organizational Process

[1*, c12s1, 2* c8; 4, part 2, part 3; 14*, c7s3]

Various aims can guide the testing phases. For example, risk-based testing uses the product risks to prioritize and focus the test strategy, and scenario-based testing defines test cases based on specified software scenarios and backlog lists. Usually, the organization of the test process includes defining test policies (i.e., specifying the purpose, goals, and overall scope of testing) and test strategies (i.e., specifying the guidelines about how testing will be carried out). For instance, in left-shift developments, a test strategy should include at least the following data: the purposes (e.g., defined through user stories), the objectives (e.g., a test suite), the scope (the SUT), and the environment and methods (e.g., how, and where the test suite is run).

5.1.3. Test Management and Dynamic Test Processes

[1*, c12; 4, part 2, part 3, 14*, c7s3]

Test activities conducted at different levels (see Section 2, Test Levels) should be organized — with people, tools, policies, and measures — into a well-defined process integral to the life cycle. Test process management includes different subprocesses such as planning, monitoring, control, and completion, whereas the Dynamic test process includes test design and implementation, test environment set-up and maintenance, test execution, and test incident reporting.

5.1.4. Test Documentation

[1*, c8s12; 14*, c7s8; 4, part 3]

According to [4, part 3], documentation is integral to the formalization of the test process. Test documents can be classified into three hierarchical categories: organizational test documentation, test management documentation and dynamic test documentation. Organizational test documentation includes the information necessary for documenting the test policy and the organizational test strategies. Test management documentation includes the test plan, test status report and test completion report. Finally,

dynamic test documentation includes the following documents: test specification (test design specification, test case specification and test procedure specification), test data requirements, test environment requirements, test data readiness report, test environment readiness report, and test execution documentation (such as actual results, test results, test execution log and incident report).

Test documentation should be produced and continuously updated with the same quality as other software engineering documentation. Test documentation should also be under the control of software configuration management. (See the Software Configuration Management KA.)

5.1.5. Test Team

[1*, c16; 2* c23s5; 4, part 2, part 3]

Formalizing the testing process may also involve formalizing the testing team's organization. Considerations of cost, schedule, maturity levels of the involved organizations and criticality of the application can guide the decision. The testing team can be composed of members involved (or not) in the SUT development (i.e., having or not having an unbiased, independent perspective) or internal (or external) personnel. Nowadays, shift-left development does not strongly distinguish among testing team members because the test suite is defined and updated according to the SUT development and delivered code.

5.1.6. Test Process Measures

[1*, c18s3, 14*, c10; 4, part 1, part 2, part 3]

Managers use several measures for the resources spent on testing, as well as for the relative fault-finding effectiveness of the various test phases, to control and improve the testing process, as well as to provide information for managing process risks. Therefore, monitor and control testing must define required data and information and state how to obtain them. The test measures may cover the number of specified, executed, passed, and failed test cases, among other elements. These measures can also be combined with specific process metrics such as residual risk, cumulative defects open and closed, test case progress, and defect detection percentage. Evaluation of test phase reports can be combined with root-cause analysis to evaluate test process effectiveness in finding faults as early as possible. Such an evaluation can be associated with risk analysis. Moreover, the resources deemed worth spending on testing should be commensurate with the application's use and criticality. Different techniques have different costs

and yield different confidence levels in product reliability.

5.1.7. Test Monitoring and Control

[4, part 1, part 2]

Monitoring and Control comprise an important subprocess of the test management process as in [4, part 2], useful for collecting data and information required during test management and assessment. Usually, monitoring and control activities are executed in parallel with the test execution, and sometimes, data collected might prompt revision of overall process planning. Monitoring assures that testing process activities comply with a specific test plan to trace the requirements satisfaction and mitigate the identified risks satisfactorily. During test monitoring and control, specific documentation (test reports) can regularly be produced to help assess and document the test activity.

5.1.8. Test Completion

[14*, c7s11; 4, part 3]

A decision must be made about how much testing is enough and when a test stage can be completed. Therefore, the purpose of *Test Completion*, a subprocess of the test management process as in [4, part 2], is to ensure that test requirements are satisfied and verified, test reports are completed, and test results are communicated to relevant stakeholders. Thoroughness measures, such as achieved code coverage or functional coverage, and estimates of fault density or operational reliability, provide useful support but are not sufficient in themselves. The decision also involves considerations about the costs and risks incurred by possible remaining failures, as opposed to the costs incurred by continuing to test (See Test Selection and Adequacy Criteria in Section 1.2, Key Issues.) As for the other activities, in this stage, specific documentation is produced (e.g., test completion report) and communicated to the relevant stakeholders.

5.1.9. Test Reusability

[14*, c3; 9]

It is necessary to add complexity and time for test planning and design to achieve reusability of the testing artifacts, such as the test case or execution environment, which is desired when test development is costly, time-consuming, and complex.

Test reusability collects and classifies the testing knowledge (test cases and test results) to make this information searchable and usable for creating new tests or re-executing an existing one. Suitable

knowledge-based repositories should be configured and managed to test reusability so changes to software requirements or design can be reflected in changes to the tests.

Currently, the reusability of test cases is pivotal in feature-based or product-line development and regression testing. Test reusability also relates to maintainability because reusability can reduce the cost and effort involved and improve a test's effectiveness.

5.2. Test Sub-Processes and Activities

[1*, c1s12; 1*, c12s9; 4, part 2]

In the remainder of this section, the main testing activities and sub-processes are briefly introduced.

5.2.1. Test Planning Process

[1*, c12s1, c12s8; 11; 4, part 2]

Like all other aspects of project management, testing activities must be planned. According to [4, part 2], key aspects of test planning include identification and coordination of personnel, identification of the test objective and completion criteria, definition of test facilities and equipment, creation and maintenance of all test-related documentation, and risk planning and management for possible undesirable outcomes. These activities can be organized at three different levels: (1) process management (i.e., identification of test policies, strategies, processes, and procedures), (2) organizational management (i.e., definition of the test phase, test type and test objective), and (3) design and implementation (i.e., definition of the test environment, the test execution process and monitoring, the completion process, and reporting).

5.2.2. Test Design and Implementation

[1*, c12s1, c12s3; 11]

Generation of test cases is based on the level of testing to be performed and the chosen testing techniques. According to the dynamic test process, as described in [4, part 2], preconditions of the test case generation are the identification of test objectives and the selection of the appropriate testing/demonstration techniques. Test generation focuses on implementing and executing test cases. It often relates to tooling (i.e., using specific software, also called a *test cases generator*). This software accepts inputs (such as source code, test criteria, specifications, or data structure definitions) and uses them to generate the test suites. Sometimes, a test case generator can determine expected results by using a specific oracle facility. This contributes to the full test automation of the overall testing process.

5.2.3. Test Environment Set-up and Maintenance

[1*, c12s6; 2* c8s1; 14* c13s2; 4, part 2; 11]

According to the dynamic test process, as described in [4, part 2], test environment development and setup involve identifying the testing infrastructure. This includes selecting or developing the facilities, hardware, software, firmware, and procedures to conduct the testing activity. The testing environment can be simulated, controlled, and executed *in vitro* or *in vivo*. Developing the test environment also involves setting up monitoring and logging facilities useful for documenting the testing activities and assessing the result obtained. The testing environment should be compatible with the other software engineering tools used.

5.2.4. Controlled Experiments and Test Execution

[1*, c12s7, 14* c4s7, 14* c5s6; 4, part 2]

Execution of tests should embody a basic principle of scientific controlled experimentation — everything done during testing should be performed and documented specifically and clearly enough that another person could replicate the results. Hence, testing should be performed following documented procedures using a clearly defined version of the SUT. Especially during acceptance testing, controlled experiments like A/B testing can also be performed to statistically evaluate user preferences between different versions of the SUT.

5.2.5. Test Incident Reporting

[1*, c13s4, c13s9, c13s11; 2*, c8s3; 14*, c7s8; 4, part 3; 12]

According to the dynamic test process, as described in [4, part 2], testing incidents and reporting focus on the well-defined test data collection process (i.e., identifying when a test was conducted, who performed the test, what software configuration was used, and other relevant identification information). This process and the collected evidence can be leveraged for accountability purposes. Test reporting can involve suitable audit systems to identify unexpected or incorrect test results and record them in a problem reporting system. These data form the basis for later debugging and fixing the problems observed as failures during testing. Also, anomalies not classified as faults could be documented if they later become more serious than first thought. Test reports are also inputs to the change management request process. (See Software Configuration Control in the Software Configuration Management KA.)

Hence, the *Test Incident Reporting* process focuses on identifying the relevant stakeholders' incidents that could be used to determine what aspects of software testing and other processes need improvement and how effective previous approaches have been.

Part of the incident reporting is also evaluating test results to determine whether the testing has been successful. In most cases, “successful” means that the software performed as expected and did not have any major unexpected outcomes. Not all unexpected outcomes are necessarily faults; sometimes they are determined to be simply noise. Before a fault can be removed, an analysis and debugging effort is needed to isolate, identify, and describe it. When test results are particularly important, a formal review board may be convened to evaluate them.

5.3. Staffing

[1*, c16; 4, part 3]

According to [4, part 3], staffing includes defining roles, activities, and responsibilities, specifying hiring needs, and defining training needs. Staffing affects project risk because the team’s expertise might undermine the ability to discover faults, to address changing requirements, to meet deadlines, and increase/reduce maintenance costs.

The roles, activities and responsibilities definition establishes the following roles and responsibilities: the activity leader and supporting personnel, the test-related roles and their corresponding responsibilities, and the person responsible for providing the test item(s).

Depending on the development lifecycle adopted, typical testing roles include but are not limited to scrum master/test lead, QA/test analyst, test designer, test security/performance engineer and consultant, test environment expert, test executor and test automation consultant or architect.

Hiring needs require the identification of specific requirements for which additional testing personnel are needed to complete the testing process (as well as when that personnel is needed and the desired skills). Depending on the business needs, staffing could take different forms, from internal transfers to external hires or even consultants and/or outsourced resources.

Finally, the training needs specification includes the definition of the required skill level. It also includes the specification of the training activities (such as classroom training, self-paced training, computer-based training, or mentoring) useful for providing the necessary skills to the selected staff.

6. Software Testing in the Development Processes and the Application Domains

[2*, c8, c15; 14*, c4s8, c7]

Whatever development process is adopted, testing remains a fundamental activity. However, specific testing activities or terminologies could be used in some cases, such as the adopted development life cycle and/or the application domain

6.1. Testing Inside Software Development Processes

[2*, c8; 14*, c7]

In the remainder of this section, peculiarities of testing inside the different development processes are provided.

6.1.1. Testing in Traditional Processes

[1* c18; 14*, c7]

There are a variety of traditional processes, essentially based on the SUT development principles, that can be adopted within the organization. Sequential, V, spiral model and iterative are just some of the processes commonly applied. (Software Life Cycles in the Software Engineering Process KA provides a detailed description of each.) However, in all these processes, testing is just one perceived activity; it is sometimes performed at the end of the process, with a tangible risk of SUT development failure in case of deviation of the end-user needs or assessment issues. During recent years, to evaluate and control the overall quality of the SUT, initiatives such as Test Maturity Model integration (TMMi) and Software Process Improvement (SPI) have been established. As a result, different existing frameworks have been updated or improved for the purpose, such as Software Process Improvement and Capability Determination (SPICE), Capability Maturity Model Integration (CMMI), and Rational Unified Process (RUP).

For instance, CMMI is one of the most referenced models; it can guide key SUT stakeholders in gaining control of their development and maintenance processes. It is, in fact, a well-defined set of best practices in software testing that improves SUT quality by increasing customer satisfaction.

Presented in the early 2000s, the RUP model can be seen as a predecessor of the left-shift movement. RUP encourages testing early by offering several mechanisms to integrate testing more closely with the software development effort, making testing a distinct discipline. Furthermore, RUP promotes an iterative development approach for continuously verifying quality. It also enables use cases and risk to drive SUT development and allows strategic change management. Indeed, RUP groups the SUT increments and SUT iterations into four phases: inception, elaboration, construction, and transition.

Nowadays, RUP can be considered both Iterative and Agile — Iterative because all the core activities are repeated throughout the SUT development project, and Agile because the defined phases of the chosen lifecycle can be repeated until the SUT meets requirements (both functional and non-functional), achieves the defined objectives, and guarantees the target quality.

6.1.2. Testing in Line with Left-Shift Movement

[2*, c3, c8s2; 4, part 1; 10, c3, c5]

The *left-shift testing movement* promotes the adoption of testing in the early stages of software development to detect and remove faults as early as possible to increase overall SUT quality and reduce the cost and risks of testing activities. Currently, different development life cycles, such as Agile, DevOps and TDD, belong to the left-shift movement. (See Agile Methods in the Software Engineering Process KA.)

In left-shift-based development, different testing aspects should be considered:

- A. The internal code quality: Regression, prioritization, security, and privacy could be the primary objectives of the internal code quality (Section 2.2). Usually, unit testing and integration testing are the targeted levels (Section 2.1), whereas structure-based is the main testing technique (Section 3.2).
- B. Business needs: Compliance and conformance, usability, security, and privacy are just a subset of the possible objectives of the business needs aspect (Section 2.2). Concerning this aspect, testing focuses more on the system and acceptance test levels and on end-user expectations, as well as usage-based (Section 3.5) and scenario-based techniques (Section 3.1.8).
- C. Perceived quality: Alpha, beta, installation, usability, security, and privacy could be the primary objectives of the internal perceived quality (Section 2.2). Perceived quality usually focuses on the acceptance test level and is achieved by applying techniques based on software engineering's intuition and experience (Section 3.3) and usage-based and fault-based techniques, such as mutation testing (Section 3.4).
- D. Quality assurance: Performance installations, security, and privacy conformance and compliance are some main objectives of quality assurance (Section 2.2). This aspect may involve all testing levels, and the selection of the testing

technique depends on the objective and the level chosen.

Here, some examples of testing inside the different left-shift movements implementation are provided:

- In Agile process development, testing activities involve all stakeholders (such as customers and team personnel) and target the identification of where improvements could be made in future interactions. Managing the risk of regression defects, meeting changing requirements, and managing their impact on test artifacts are also objectives of the Agile testing process. Typically, test automation is used to manage the regression risk, and exploratory testing may be used to manage a lack of detailed requirements.
- In TDD, the test cases mainly target the software requirements specifications and acceptance, and they are generated in advance of the code being written. The tests are based on the user stories and implemented using automated component testing tools. Indeed, TDD is a practice that requires defining and maintaining unit tests and can help clarify the user needs and software requirements specifications.
- In testing automated builds and continuous integration (for instance, DevOps), the SUT is continuously developed, integrated, delivered and monitored. In this process, regression testing is continuously performed to timely identify and correct development and integration issues. Additionally, quick testing techniques, such as smoke testing, are commonly used during continuous integration to guarantee that the SUT is testable before it is released to the operational stage.

6.2. Testing in the Application Domains

[2*, c15; 14*, c4s8]

Usually, an application domain is strictly connected to a certain reality. Therefore, testing approaches could be tailored to the needs of the domain and customized to the adopted technologies.

Below, we provide an overview of different aspects and solutions for software testing applied within several domain-specific environments:

- Automotive domain testing: Due to the complexity of automotive systems, this testing involves aspects of almost every software component and its interaction with hardware. Security testing, simulation testing, reliability/life cycle testing, integrated systems

- testing, data acquisition and signal analysis testing, quality testing and inspection, and stress/strain testing are just some of the various testing performed in this domain. Several supporting standards are currently available to guide and manage automotive testing according to the peculiarity, the component, or the quality aspect that should be assessed. Autosar² and Automotive SPICE³ are examples.
- Internet of Things (IoT) domain testing: This testing involves application development, device management, system management, heterogeneity management, data management, and tools for analysis, deployment, monitoring, visualization and research. Additionally, security, privacy, communications and user/component interaction should be considered in the quality assessment. For example, guidelines and specific conformance test suites for cybersecurity assessment of the IoT SUT are detailed in the European Telecommunications Standards Institute (ETSI) standards.⁴
 - Legal domain testing: One of the most important aspects in the legal domain is the management of highly sensitive users; therefore, security, privacy and trust are the most common areas of focus for testing. Additionally, because of the copious data collected and exchanged, performance testing of the data repository, testing to show accurate communication and integration testing, as well as consistency and compliance testing, should also be done. Finally, because the legal domain is characterized by specific nomenclature and jargon, involving legal domain experts in test case generation is common practice to ensure a focus on desired characteristics and quality.
 - Mobile domain testing: This testing is usually for usability, functional, configuration and consistency assessment. Mobile-specific aspects such as screen resolution, Global Positioning System (GPS), operating systems, and device manufacturers should also be considered during testing activity. Finally, the type of mobile applications (native or web apps) and their interactions need to be tested. For example, the W3C Web and Mobile Interest Group⁵ provides facilities, guidelines and ad hoc test suites useful for developing and testing web-based content, applications and services.
 - Avionics domain testing⁶: Usually, avionics systems include several independent or loosely coupled components and commercial off-the-shelf products. Those forces testing to include very general processes and approaches applicable at both the system and the process levels. Functional and non-functional, integration, communication operational, stress, safety, and security testing are examples of possible approaches. As in other domains, supporting standards such as Aeronautical Radio Incorporated (ARINC) Standards and ASTM F3153-15 can be used for reference.
 - Healthcare domain testing: Healthcare domain testing should ensure quality in areas such as secure and reliable data exchange, stable performance, privacy, and safety. Interoperability, usability, performance and compliance with industry regulations, as well as security and safety standards (such as the Health Level Seven (HL7),⁷ Fast Healthcare Interoperability Resources (FHIR),⁸ Digital Imaging and Communications in Medicine (DICOM),⁹ Health Insurance Portability and Accountability Act (HIPAA),¹⁰ and the General Data Protection Regulation (GDPR)¹¹) should also be considered.
 - Embedded domain testing: Because software and hardware are tightly coupled in embedded systems, testing activity should assess functional and non-functional attributes of both software and hardware.
 - Graphical User Interface (GUI) testing: GUI testing involves assessing the UI (User Interface) (i.e., the elements of the user objects that we can see). Thus, GUI testing targets the design pattern, images, alignment, spellings, and the overall look and feel of the UI. Testing approaches based on finite-state machines, goal-driven approaches, approaches based on abstractions and model-based approaches can be considered.
 - Gaming: Gaming applications and software are currently a very active sector of software production, causing increased demand for new approaches and ways to ensure their quality and security. Among the specific testing techniques,

² <https://www.autosar.org/>

³ <https://www.automotivespice.com/>

⁴ <https://www.etsi.org/>

⁵ <https://www.w3.org/2013/07/webmobile-ig-charter.html>

⁶ www.astm.org

⁷ <https://www.hl7.org/>

⁸ <http://fhir.org/>

⁹ <https://www.dicomstandard.org/>

¹⁰ <https://www.hhs.gov/hipaa/>

¹¹ <https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679>

playtesting is one of the most adopted. In this case, real gamers repeat quality control methods at many points of the game execution or design process. GUI testing, functionality testing, security testing, console testing, compliance testing and performance testing can also be considered.

- Real-time domain testing: Real-time testing usually focuses on assessing timing constraints and deterministic behavior. Usually, unit, integration and system testing approaches can be adopted. Communication, interaction and behavioral testing can also be performed.
- Service Oriented Architecture (SOA) testing: This testing focuses mainly on correctly implementing business processes and involves unit and integration testing approaches. Structure-based, specification-based and security testing can be applied. The testing activity might vary according to the environment, organization and set of requirements that should be satisfied.
- Finance domain testing: This testing covers a wide range of aspects, from managing financial requirements to assessing financial applications and software programs. As in other domains, domain-specific knowledge (such as that held by, for example, banks, credit unions, insurance companies, credit card companies, consumer finance businesses, investment funds and stock brokerages) could be necessary to apply the testing process effectively and efficiently. Customer satisfaction, usability, security, privacy, third-party component and apps integrations, real-time issues, and performance are some of the most important challenges in this domain.

7. Testing of and Testing Through Emerging Technologies

In recent decades, software development was driven by emerging trends such as the widespread diffusion of mobile technology, cloud infrastructures adoption, big data analysis and the software as a service paradigm, which highlighted new constraints and challenges for testing.

7.1. Testing of Emerging Technologies

- Testing Artificial Intelligence (AI), ML/Deep Learning (DL) [13]: AI, ML and DL are successfully being applied in practice. Sooner or later, most business applications will have some form of AI, ML or DL. Because of their peculiarities, testing such applications is challenging and might be very expensive. AI, ML

or DL testing refers to any activity designed to reveal AI, ML or DL bugs.

- Three main aspects should be considered in defining bugs and testing in this scenario: the required conditions (correctness, robustness, security, and privacy); the AI, ML or DL items (e.g., a bug might exist in the data, the learning program, or the framework used); and the involved testing activities (test case generation, test oracle identification and definition, and test case adequacy criteria).
- In all these applications, a prototype model is first generated based on historical data. Then, off-line testing, such as cross-validation, is conducted to verify that the generated model satisfies the required conditions. Usually, after deployment, the model is used for prediction purposes by generating new data. Finally, the generated data is analyzed through online testing to evaluate how the model interacts with user behaviors.
- Testing blockchain [15]: The commonly used testing techniques for validating blockchains and related applications such as smart contracts are stress testing, penetration testing and property testing. However, depending on the specific situation, different aspects should be considered during the testing of a blockchain-based SUT, such as the following:
 - Platform type: The level of validation depends on the type of platform used for implementation — public or private. The latter requires a much greater testing effort.
 - Connection with other applications: Integration testing should be performed to check consistency when the blockchain works with various applications.
 - Performance: Performance testing should be conducted when performance issues are a concern. Specific strategies to handle many transactions should be conceived to guarantee a satisfactory performance level. Qualitative and quantitative metrics, such as average transaction validation latency and security, should also be considered.
 - Testing the cloud [1*, c10s10, 2*, c18]: Testing the cloud validates the quality of applications and infrastructures deployed in the cloud by considering both functional and non-functional properties. The focus is to identify problems posed by systems residing in the cloud. Therefore, testing activities use techniques to validate cloud-based services' performance, scalability, elasticity and security. Moreover, testing should also focus on compatibility and

interoperability among heterogeneous cloud resources when different deployment models are used (e.g., private, public or hybrid).

- Testing concurrent and distributed applications [1*, c10s10, 2*, c17]: One main aspect of testing dynamic, complex, distributed or concurrent applications is dealing with multiple operating systems and updates, multiple browser platforms and versions, different types of hardware, and many users. For such testing, it's difficult to use testing approaches based on the classical hierarchy between components or systems; instead, solutions based on input/output, dependency threads, or dynamic relations often work better. Additionally, the possibility of continuous integration and deployment of the different components forces the testing process to include approaches for managing continuous test operation, injection, monitoring and reporting according to the time, bandwidth usage, throughput, and adaptability constraints. Finally, there is still the need for solutions that allow the reusability of testing knowledge, architectures, and code to make the testing activity more effective and less expensive.

7.2. Testing Through Emerging Technologies

- Testing through ML [13]: AI, ML or DL techniques are successfully used to reduce the effort involved in several activities in software engineering (such as behavior extraction, testing or bug fixing). These techniques aid both researchers and practitioners in adopting and identifying appropriate methods for their desired applications. There is a growing interest in adopting ML techniques in software testing because most software testing issues are being formulated as ML learning problems. Indeed, AI, ML or DL is intensively used in almost all software, such as test case design, the oracle problem, test case evaluation, test case prioritization and refinement, and mutation testing automation. Indeed, they can reduce maintenance efforts and improve the overall SUT quality because of their ability to analyze large amounts of data for classifying, triaging and prioritizing bugs more efficiently. From a DevOps perspective, AI, ML and DL solutions can be used in SUT automation authoring and execution phases of test cases, as well as in the post-execution test analysis that identifies trends, patterns and impact on SUT testing activity.
- Testing through blockchain [15]: Testing becomes complicated when different teams, domain experts and users need to work together in collaborative, large-scale systems and complex

software systems to achieve a common goal. This is mainly because of the time constraint, data sharing policies, acceptance criteria and trusted coordination among the teams involved in the testing process. Blockchain technologies can be exploited to improve software testing efficiency and avoid using centralized authority to manage different testing activities. This can help ensure distributed data management, tamper resistance, auditability, and automatic requirement compliance to improve the quality of software testing and development. Blockchain-based approaches for trusted test case repository management and to support test-based software and security testing are also considered.

- Testing through the cloud [17]: Testing through the cloud refers to SUT testing performed by leveraging scalable cloud technologies. Usually, the cloud is used for testing purposes wherever large-scale simulations and elastic resources are necessary. Indeed, this can affect cost reduction, development, and maintenance of the testing infrastructure (scaffolding), and online validation of systems, such as ML-based SUT. A particular situation is the testing of the cloud through the cloud itself. This is an example of the intersection between testing of and testing through emerging technologies. The applications and infrastructures deployed in the cloud can be tested, exploiting the cloud's bandwidth.
- Testing through simulation [1*, c3s9]: Simulation is an important technology for testing activity because it represents a valid means for evaluating SUT execution under critical situations or disasters or assessing specific behaviors or recovering activities. The complexity of the testing approach might vary according to the complexity of the simulation system adopted and might involve closed-loop testing; assessing the devices, communications, and interface; and use of real-time data (e.g., voltage, current and breaker status). Simulation testing can be applied to each development level and might involve mathematical, formal representation of the real system, environment, network conditions and control devices. Simulation testing is currently adopted in many application domains. Especially in the automotive and embedded domain, among the different proposals, one of the emerging solutions for simulation testing is hardware-in-the-loop (HIL) simulation testing. In this case, real signals sent to the SUT to simulate reality and to test and design the iteration are continuously performed while the real-world system is being used.

- Testing through crowdsourcing [16]: Crowdsourced testing (also known as crowdtesting) is an emerging approach for involving users and experts in the testing activity. Thus, crowdsourcing uses represent the dispersed, temporary workforce of multiple individual testers. Testing through crowdsourcing is mainly used for testing mobile applications because it ensures technology diversity and customer-centric validation. However, crowdtesting is not a substitute for in-house SUT validation. It represents a valid means of detecting failures and issues because it involves many individuals (testers) in different locations, who are using different technologies in different conditions and who have different skills and knowledge.

8. Software Testing Tools

[1*, c12s11, 14*, c7]

Several testing tools focus on the SUT peculiarities and needs. This section describes the main issues and challenges concerning testing tools and provides an overview of their currently identified categories.

8.1. Testing Tool Support and Selection

[1*, c12s11, 14*, c7]

Testing involves many labor-intensive tasks since it involves running numerous program executions and handling a considerable amount of information. Appropriate tools can alleviate the burden of tedious clerical operations and make them less error-prone. Sophisticated tools can support test design and generation, making them more effective.

Guidance to managers and testers on selecting testing tools that will be most useful to their organization and processes is an important topic, as tool selection greatly affects testing efficiency and effectiveness. Tool selection depends on diverse factors, such as development choices, evaluation objectives and execution facilities. In general, there might not be a unique tool to satisfy specific needs, so a suite of selected tools could be appropriate.

8.2. Categories of Tools

[1*, c1, c3, c4, c7, c8, c9, c12]

Several classifications of testing tools mainly describe their functionalities, such as the following:

- *Test harnesses* (drivers, stubs) [1*, c3s9] provide a controlled environment in which tests can be launched and the test outputs can be logged. Drivers and stubs are provided to execute parts of a SUT to simulate calling and called modules.

- *Test generators* [1*, c12s11] assist in generating test cases. That generation can be random, path-based, model-based or a mix thereof.
- *Capture/replay tools* [1*, c12s11] automatically re-execute or replay previously executed tests that have recorded inputs and outputs (e.g., screens).
- *Oracle/file comparators/assertion checking tools* [1*, c9s7] assist in deciding whether a test outcome is successful.
- *Coverage analyzers and instrumenters* [1*, c4] work together. Coverage analyzers assess which and how many entities of the program flow graph have been exercised among all those required by the selected test coverage criterion. The analysis can be done through SUT instrumenters that insert recording probes into the code.
- *Tracers* [1*, c1s7] record the history of a program's execution paths.
- *Regression testing tools* [1*, c12s16] support the re-execution of a test suite after a section of software has been modified. They can also help select a test subset according to the change made.
- *Reliability evaluation tools* [1*, c8] support test results analysis and graphical visualization to assess reliability-related measures according to selected models.
- *Injection-based tools* [1*, c3, c7s7] focus on introducing or reproducing specific problems to confirm that the SUT behaves suitably under the corresponding condition. That can involve managing some input or triggering of events. Usually, two categories of injection-based tools are considered: attack injection and fault injection.
- *Simulation-based tools* [1*, c3s9] verify and validate selected properties. Usually, they exploit specific models to enable the automated execution of scenarios to assess whether the SUT operates as expected or to predict how the SUT would respond to defined inputs. Typical simulation-based tools are classified into tools for verification, tools for collaboration, tools for optimization, tools for testing automated systems and tools for evaluating software concepts.
- *Security testing tools* [1*, c8s3, c12s11] focus on specific security vulnerabilities. Among these are tools for attack injection, penetration testing and fuzzy testing.
- *Test management tools* [1*, c12s11] include all the supporting tools that assure efficient and effective test management and data collection.
- *Cross-browser testing tools* [1*, c8s3] enable the tester to quickly build and run user interface test

- cases across desktop, mobile and web applications to check whether the SUT looks and works as expected on every device and browser.
- *Load testing tools* [1*, c3] collect valuable data and evidence for SUT performance evaluations.
- *Defect tracking tools* [1*, c3] help keep track of detected faults during the SUT development projects. These tools behave as tracking systems and usually allow end users to enter fault reports directly.
- *Mobile testing tools* [1*, c8s3] support the implementation and testing of mobile apps by allowing several repeated UI tests over the application platform, development on real mobile devices or emulators, testing of the mobile apps on real-time implementations and collection of data for specific QA measures.
- *API testing tools* [1*, c7s2] check whether the applications meet functionality, performance,

reliability, and security expectations throughout the automation of specific API tests.

- *CSS validator tools* [1*, c7s2] validate Cascading Style Sheets (CSS) code and discover errors, issues and warnings that can be fixed. The CSS Validation Service, provided by W3C for free, is one of the most used validators in practice that helps both web designers and web developers check CSS.
- *Web application testing tools* [1*, c8s3], also referred to as web testing tools, support validating the functionality and the performance of web-based SUTs before their deployment into production. These tools provide relevant insight and data for different stakeholders, such as developers, servers, and infrastructure administrators. From a DevOps perspective, these tools address issues, or bugs before SUTs are available to end users.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	1*	2*	14*	19*
1. Software Testing Fundamentals	c1, c2	c8	c7	
<i>1.1. Faults vs. Failures</i>	c1s5	c1	c1s3	
<i>1.2. Key Issues</i>				
1.2.1. Test Case Creation	c12s1, c12s3	c8		
1.2.2. Test Selection and Adequacy Criteria	c1s14, c6s6, c12s7	c8		
1.2.3. Prioritization/Minimization				
1.2.4. Purpose of Testing	c13s11, c11s4	c8		
1.2.5. Assessment and Certification		c7, c25		
1.2.6. Testing for Quality Improvement/Assurance	c16s2			
1.2.7. The Oracle Problem	c1s9, c9s7			
1.2.8. Theoretical and Practical Limitations	c2s7			
1.2.9. The Problem of Infeasible Paths	c4s7			
1.2.10. Testability	c17s2			
1.2.11. Test Execution and Automation				
1.2.12. Scalability	c8s7			
1.2.13. Test Effectiveness	c1s1	c8s1		
1.2.14. Controllability, Replication and Generalization	c12s12			
1.2.15. Off-line vs. Online Testing				
<i>1.3. Relationship of Testing to Other Activities</i>				
2. Test Levels	c1s13	c8s1		
<i>2.1. The Target of the Test</i>	c1s13	c8s1		
2.1.1. Unit Testing	c3	c8		
2.1.2. Integration Testing	c7	c8		
2.1.3. System Testing	c8	c8		
2.1.4. Acceptance Testing	c1s7	c8s4		
<i>2.2. Objectives of Testing</i>	c1s7			
2.2.1. Conformance Testing	c10s4			
2.2.2. Compliance Testing	c12s3			

2.2.3. Installation Testing	c12s2			
2.2.4. Alpha and Beta Testing	c13s7, c16s6	c8s4		
2.2.5. Regression Testing	c8s11, c13s3			
2.2.6. Prioritization Testing	c12s7			
2.2.7. Non-functional testing	c8s7, c8s8, c14s2, c15, c8, c11, c17 c17s2			
2.2.8. Security Testing		c13		
2.2.9. Privacy Testing		c13, c14		
2.2.10. Interface and API Testing		c8s1	c7s12	
2.2.11. Configuration Testing	c8s5			
2.2.12. Usability and Human-Computer Interaction Testing		c8s4		c6
3. Test Techniques	c1s15			
<i>3.1. Specification-Based Techniques</i>	c6s2			
3.1.1. Equivalence Partitioning	c9s4			
3.1.2. Boundary-Value Analysis	c9s5			
3.1.3. Syntax Testing	c10s11	c5		
3.1.4. Combinatorial Test Techniques	c9s3			
3.1.5. Decision Table	c9s6, c13s6			
3.1.6. Cause-Effect Graphing	c1s6			
3.1.7. State Transition Testing	c10			
3.1.8. Scenario Testing		c8s3.2, c19s3.1		
3.1.9. Random Testing	c9s7			
3.1.10. Evidence-Based				
3.1.11. Forcing Exception				
<i>3.2. Structure-Based Test Techniques</i>				
3.2.1. Control Flow Testing	c4			
3.2.2. Data Flow Testing	c5			
3.2.3. Reference Models for Structure-Based Test Techniques	c4			
<i>3.3. Experience-Based Techniques</i>				

3.3.1. Error Guessing	c9s8			
3.3.2. Exploratory Testing				
3.3.3. Further Experience-Based Techniques				
3.4. Fault-Based and Mutation Techniques	c1s14, c3s5			
3.5. Usage-Based Techniques	c15s5			
3.5.1. Operational Profile	c15s5	c11		
3.5.2. User Observation Heuristics				c5, c7
3.6. Techniques Based on the Nature of the Application		c16, c17, c18, c4s8 c20, c21		
3.7. Selecting and Combining Techniques			c7s12	
3.7.1. Combining Functional and Structural	c9			
3.7.2. Deterministic vs. Random	c9s6			
3.8. Techniques Based on Derived Knowledge		c19, c20	c7	
4. Test-Related Measures		c24s5	c10	
4.1. Evaluation of the SUT		c24s5		
4.1.1. SUT Measurements That Aid in Planning and Designing Tests			c10	
4.1.2. Fault Types, Classification and Statistics	c13s4, c13s5, c13s6			
4.1.3. Fault Density	c13s4		c10s1	
4.1.4. Life Test, Reliability Evaluation	c15	c11	c1s3	
4.1.5. Reliability Growth Models	c15	c11s5		
4.2. Evaluation of the Tests Performed				
4.2.1. Fault Injection	c2s5			
4.2.2. Mutation Score	c3s5			
4.2.3. Comparison and Relative Effectiveness of Different Techniques	c1s7			
5. Test Process		c8		
5.1. Practical Considerations				
5.1.1. Attitudes/Egoless Programming	c16	c3		
5.1.2. Test Guides and Organizational Process	c12s1	c8	c7s3	

5.1.3. Test Management and Dynamic Test Processes	c12		c7s3	
5.1.4. Test Documentation	c8s12		c7s8	
5.1.5. Test Team	c16	c23s5		
5.1.6. Test Process Measures	c18s3		c10	
5.1.7. Test Monitoring and Control				
5.1.8. Test Completion			c7s11	
5.1.9. Test Reusability			c3	
<i>5.2. Test Sub-Processes and Activities</i>	c12s9, c1s12			
5.2.1. Test Planning Process	c12s1, c12s8			
5.2.2. Test Design and Implementation	c12s1, c12s3			
5.2.3. Test Environment Set-up and Maintenance	c12s6	c8s1	c13s2	
5.2.4. Controlled Experiments and Test Execution	c12s7		c4s7, c5s6	
5.2.5. Test Incident Reporting	c13s4, c13s9, c13s11	c8s3	c7s8	
<i>5.3. Staffing</i>	c16			
6. Software Testing in the Development Processes and the Application Domains		c8, c15	c4s8, c7	
<i>6.1. Testing Inside Software Development Processes</i>		c8	c7	
6.1.1. Testing in Traditional Processes	c18		c7	
6.1.2. Testing in Line with Left-Shift Movement		c3, c8s2		
<i>6.2. Testing in the Application Domains</i>		c15	c4s8	
7. Testing of and Testing Through Emerging Technologies				
<i>7.1. Testing of Emerging Technologies</i>	c10s10	c17, c18		
<i>7.2. Testing Through Emerging Technologies</i>	c3s9			
8. Software Testing Tools	c12s11		c7	
<i>8.1. Testing Tool Support and Selection</i>	c12s11		c7	
<i>8.2. Categories of Tools</i>	c1, c3, c4, c7, c8, c9, c12			

References

- [1*] Naik, S., and Tripathy P., *Software Testing and Quality Assurance: Theory and Practice*, ed: Wiley, 2008, p. 648.
- [2*] Sommerville, I., *Software Engineering*, 10th ed., Addison-Wesley, 2016.
- [3] Dijkstra, E.W., *Notes on Structured Programming*, Technological University, Eindhoven, 1970.
- [4] ISO/IEC/IEEE 29119 — System and software engineering — Software testing, ed. 2021.
- [5] ISO/IEC/IEEE International Standard — Systems and software engineering — Vocabulary, in *ISO/IEC/IEEE 24765:2017(E)*, Aug. 28, 2017, pp. 1-541.
- [6] Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y., and Harman, M., Chapter Six — Mutation Testing Advances: An Analysis and Survey, *Adv. Comput.* 112, 2019: 275-378.
- [7] Utting, M., Legeard, B., Bouquet, F., Fournieret, E., Peureux, F., and Vernotte, A., Recent advances in model-based testing, *Advances in Computers*, 101, 2016, pp. 53-120.
- [8] IEEE Std 1012-2016, IEEE Standard for System, Software, and Hardware Verification, and Validation, ed. 2016.
- [9] ISO/IEC 25010:2011, Systems and software engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE) — System and Software Quality Models, ed. 2011.
- [10] IEEE 2675-2021, IEEE Standard for DevOps: Building Reliable and Secure Systems Including Application Build, Package, and Deployment, ed. 2021.
- [11] Software Engineering Competency Model (SWECOM), v1.0, 2014.
- [12] ISO/IEC 20246:2017, “Software and systems engineering — Work product reviews”, ed, 2017, 42p
- [13] Riccio, V., Jahangirova, G., Stocco, A., et al., Testing machine learning based systems: A systematic mapping, *Empir Software Eng*, 25, 2020, pp. 5193-5254.
- [14*] Laporte, C.Y., and April, A., *Software Quality Assurance*, IEEE Computer Society Press, 1st ed., 2018.
- [15] Demi, S., Colomo-Palacios, R., and Sánchez-Gordón, M., Software Engineering Applications Enabled by Blockchain Technology: A Systematic Mapping Study, *Applied Sciences*, 11(7), 2021, pp. 2960.
- [16] Mao, Ke, Capra, Licia, Harman, Mark, and Jia, Yue. A survey of the use of crowdsourcing in software engineering, *Journal of Systems and Software*, 126, 2017, pp. 57-84.
- [17] Bertolino, A., Angelis, G.D., Gallego, M., García, B., Gortázar, F., Lonetti, F., and Marchetti, E., A systematic review on cloud testing, *ACM Computing Surveys (CSUR)*, 52(5), 2019, pp. 1-42.
- [18] Achary, Rathnakar, and Pethuru, Raj, *Cloud Reliability Engineering: Technologies and Tools*, CRC Press, 2021.
- [19*] Nielsen, J., *Usability Engineering*, 1st ed., Boston: Morgan Kaufmann, 1993.

CHAPTER 6

SOFTWARE ENGINEERING OPERATIONS

ACRONYMS

API	Application Programming Interface
ATDD	Acceptance Test Driven Development
CD	Continuous Delivery
CI	Continuous Integration
CPU	Central Processing Unit
CONOPS	Concepts of Operations
DBMS	Database Management System
IaC	Infrastructure as Code
IaaS	Infrastructure as a Service
IT	Information technology
ITIL	IT Infrastructure Library
KA	Knowledge Area
KPI	Key Performance indicator
MR	Modification request
MVP	Minimum Viable Product
PaaS	Platform as a Service
PR	Problem Report
QA	Quality Assurance
SaaS	Software as a Service
SLAs	Service-Level Agreements
TDD	Test Driven Development

Software engineering operations refers to the set of activities and tasks necessary to deploy, operate and support a software application or system while preserving its integrity and stability. These activities include the deployment and configuration of the software in the targeted operational environments and the monitoring and management of the application while it is in use (until it is retired). Once the application is operational, software engineering operations must manage any defects that are uncovered, any changes made to the system software environment and hardware equipment over time, and any new user requirements that surface.

Software engineering operations is an integral part of system and software life cycle processes [3]. The Software Engineering Operations Knowledge Area (KA) is related to all other aspects of software engineering. Therefore, this KA description is linked to all other software engineering KAs of the *SWEBOk Guide*, particularly the Software Construction KA, which discusses preparing the software for deployment, including integrating, building, packaging and testing.

Specialized software and information technology (IT) operations engineers have traditionally provided and managed IT operations services. Best practices in software engineering operations were initially published by the IT Infrastructure Library (ITIL) and were quickly accepted by the industry. These practices were summarized and published in the Institute of Electrical and Electronics Engineers 20000 standard [1].

INTRODUCTION

Historically, operations and computing centers were often located in organizational

silos separate from software development activities. Progressive organizations now co-locate software development, software maintenance and some software engineering operations activities (often provided as a service). Benefits of this approach are the elimination of the organizational silos that separated these software activities and the sharing of common processes and tools. The rising popularity and growing acceptance of DevOps practices [2*] and related standards [4], including an ever-evolving set of tools, reflect this trend. DevOps aims at automating and continuously evolving software engineering activities to ensure high-quality software and to satisfy users who demand quicker turnaround from software engineers.

The software engineer's role in software engineering operations has significantly evolved over the past decade with the emergence of DevOps, Infrastructure as Code (IaC), Agile infrastructure practices, and the availability of Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) solutions. Tasks traditionally performed by IT infrastructure engineers are increasingly automated and made available as a service, enabling application developers to perform software engineering operations tasks independently as part of

their daily project activities. For example, application developers in many organizations can now directly use IaaS and PaaS to deploy applications in production environments and to monitor different aspects of those applications without directly involving operations engineers.

Although many organizations still use conventional IT operations management processes, this KA focuses mainly on the role of software engineers in operations in the emerging contexts of DevOps, IaC and Agile infrastructure practices. For this purpose, we identify two main software engineering roles related to operations: *Operations engineer*, who is responsible for developing operations services made available as a service and accessible through an application programming interface (API), and *software engineer*, who can use the resulting operations services (available as a service) to independently deploy and manage applications without directly involving IT operations specialists.

BREAKDOWN OF TOPICS FOR SOFTWARE ENGINEERING OPERATIONS

The breakdown of topics for the Software Engineering Operations KA is shown in Figure 1.

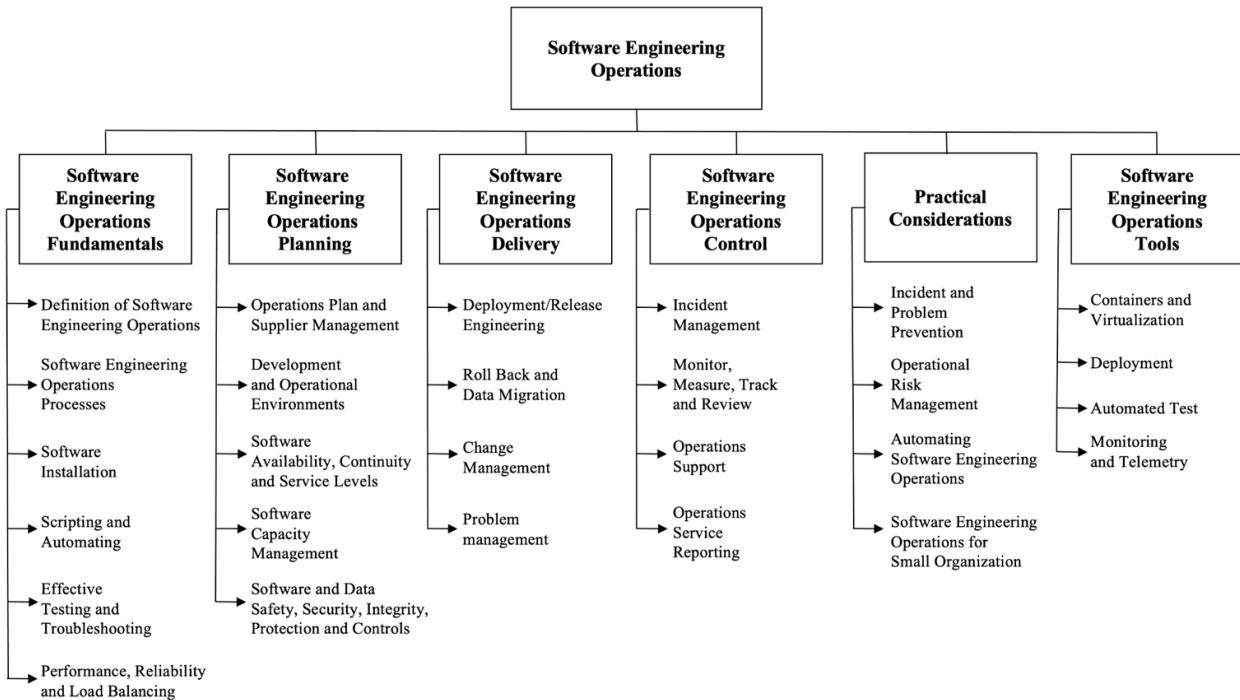


Figure 1. Breakdown of Topics for the Software Engineering Operations KA

1. Software Engineering Operations Fundamentals

1.1. *Definition of Software Engineering Operations* [1, c3s3.3][3, c6s6.4.12]

In this *Guide*, the term *software engineering operations* refers to the knowledge, skills, processes and tools used by software engineers or their organization to ensure that a software product, including IT infrastructure, system software, and application software, operates well during development, maintenance and in real conditions of operations.

In IEEE 12207 [3], an *operator* is defined as an “individual or organization that performs the operations of a system.” The *SWEBOk Guide* modifies that definition for the term *operations engineer*, which refers to a software engineer who executes software engineering operations processes. In this role, an *operations engineer* works

This first section introduces the concepts and terminology that form an underlying basis for understanding the role and scope of software engineering operations.

closely with *software engineers* to develop and offer operations services such as the following:

- Provisioning, deploying and configuring, and supporting containers and virtual servers
- Designing and offering on-demand services (e.g., environment on demand, versioning, continuous integration (CI) and testing, deployment, and surveillance) for use by software engineerins,
- Monitoring and troubleshooting system and application software incidents by running diagnostics, documenting problems and resolutions, prioritizing problems, and assessing impact of issues,
- Performing, automating and implementing appropriate processes for security, data protection and failover procedures,

- Overseeing capacity, storage planning and database management system (DBMS) performance,
- Providing documentation and technical specifications to IT staff for planning and implementing new or upgraded IT infrastructure and system software.

IEEE 20000 [1] describes the need to develop and enhance the professional competencies of operations engineers. To achieve this goal, software organizations should address the following:

- Staff recruitment: To validate job applicants' qualifications/competencies, including their professional certifications, and to identify their strengths, weaknesses and potential capabilities against the operations engineer job description, core technologies and computer languages mastered and overall experience,
- Resource planning: To staff new or expanded engineering operations services, plan the use of new technology, plan the assignment of service management staff to development project teams, develop succession planning and other staffing gaps created by staff turnover,
- Resource training and development: To identify training and development requirements and create a training and development plan that meets them; also, to provide timely, effective delivery of operations services. Operations engineers should be trained in the relevant aspects of service management (e.g., via training courses, self-study, mentoring and on-the-job training), and their teamwork and leadership skills should be developed. A chronological training record should be maintained for each

individual, with descriptions of the training provided.

1.2. *Software Engineering Operations Processes* [2*, s1][3, c6s6.4.12]

IEEE 20000 [1] is the reference standard that presents an overview of operations processes. It specifies requirements for the design, transition, delivery and improvement of operations services. The IEEE 20000 describes five main operations process groups: service delivery processes, release processes, control processes, resolution processes and relationship processes. These operations processes are further categorized as technical processes in IEEE 12207 [3]. Operations processes, from the perspective of a software engineer, contain the activities and tasks necessary to deploy, configure, operate and support an existing software system or product while preserving its integrity. The IEEE 12207 describes four main operations process activities: 1) prepare for the operation: that requires to define an operation strategy; 2) perform the operation: which consist of operating and monitoring; 3) manage the results of operation: where anomalies are recorded and addressed; and finally 4) support the customer: which means to give assistance and consultation to any user of the operations services.

Finally, IEEE 2675 [4] introduces a number of software engineering operations activities using an Agile and a minimum viable product (MVP) perspective. This standard recognizes the influence of DevOps as a set of principles and practices that enable better communication and collaboration between relevant stakeholders for the purpose of specifying, developing, continuously improving, and operating software and system products and services. These processes and activities are the responsibility of operations engineers.

For the purpose of the *SWEBOk Guide*, engineering operations activities can be grouped into three main operations processes (see Figure 2) that each contain a number of operations activities, which are described in the following sections of this chapter:

- Operations Planning (section 2)
- Operations Delivery (section 3)
- Operations Control (section 4)

- Operations Plan and Supplier Management
- Development and operational environments
- Software CM, Build, Package and Deployment
- Software Availability, Continuity and Service Levels
- Software Capacity Management
- Software Backup, Disaster Recovery and Failover
- Software and Data Safety, Security, Integrity, Protection and Controls

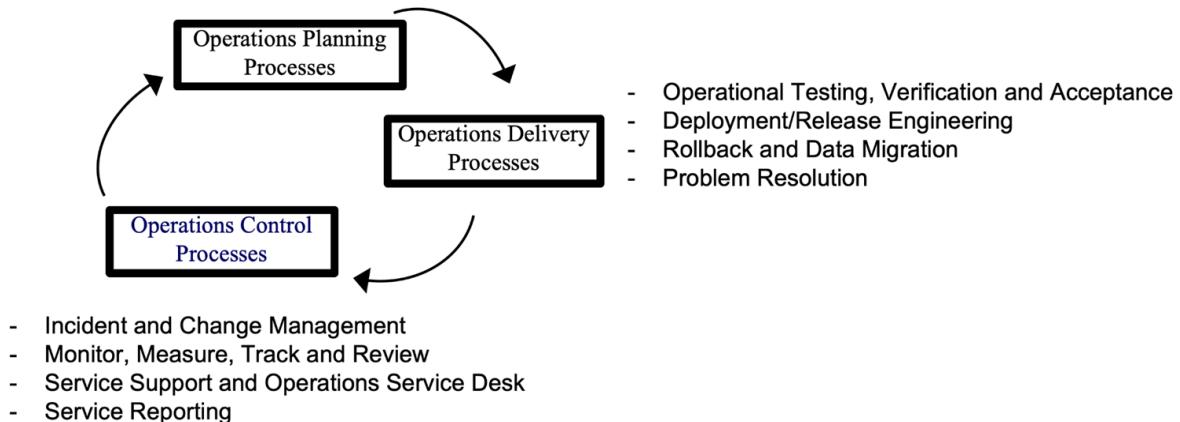


Figure 2. Software Engineering Operations Processes and Activities

Each software engineering operations process includes activities performed during the pre delivery and post delivery stages of a software project. Software engineering operations planning activities occur during the pre delivery stage. These activities are covered in this chapter.

1.3. Software Installation [1, c3, c6s2][2*, c3s3.1]

Before a software application or update can be made available to the users (i.e. released in production), the operations engineer must install the software as part of its

deployment. To install the software, the engineer might have to uninstall previous versions, configure the software for its target destination, and create the necessary directories, registry files and environment variables on the target destination. This is often done using a scripting language. The installation of the software to the appropriate locations is typically done electronically, but in the case of embedded systems, it might require the use of a

- Operational Testing, Verification and Acceptance
- Deployment/Release Engineering
- Rollback and Data Migration
- Problem Resolution

physical medium. Once the software is installed, a verification step is conducted to ensure that the operation succeeded.

1.4. Scripting and Automating [2*, c9]

As part of software engineering operations, repetitive tasks are automated to reduce delays, increase quality, and ensure a consistent and stable operational environment. This is typically achieved using scripting languages, which are basic programming languages. Automating operations enables a quicker reaction in case of a failure and, therefore, results in less downtime and fewer severe incidents, as alerts are sent immediately. Automating such tasks is also a good way to ensure

standardization of operations in an organization. It also constitutes the basis for the development of operations made available as a service. Refer to section 6 for further discussion on operations tools.

1.5. Effective Testing and Troubleshooting

[2*, c3]

Software engineering operations is responsible for ensuring the stability of the system. For this purpose, software must be thoroughly tested before it is released (deployed in production and made available to users). Because manual testing is inefficient, error-prone and non-scalable, testing must be automated as much as possible throughout the entire software process. Also, because the time available for testing is limited, regression testing and test coverage strategies (the selective retesting of a software application, or component, to verify that the software to be deployed will not cause unintended effects) play an important role in software engineering operations.

When errors are found (in production after the software is released or during internal testing phases), software operations engineers need to troubleshoot hardware and software incidents by running diagnostics, documenting problems and resolutions, prioritizing problems, and assessing the impact of the issues. The cost — in both time and money — of repeating full testing on a major piece of software is significant. To ensure that the requested problem reports (PRs) are valid, the operations engineer should replicate and verify problems by running the appropriate tests. Testing certain aspects of the software in production can be particularly challenging. For example, when software performs critical functions, bringing it off-line to test might be difficult. Generally, testing the software in the production system context is challenging (sometimes

impossible) and could require the use of testing techniques such as canary testing and dark launches. The Software Testing KA provides additional information and references on testing.

1.6. Performance, Reliability and Load Balancing

[1, c6s6.2]

Software operations engineers plan for performance, reliability and load balancing early in software projects to ensure they meet the project requirements. (See section 1.2 to 1.7 of the Software Requirements KA). A current trend is for software engineers to design and use infrastructure/operations services to adjust dynamically (e.g. scalability) the infrastructure according to the demand. Using DevOps practices enables operations engineers to anticipate these needs early and provide infrastructure services that software engineers can use and test during the development stages of a project.

2. Software Engineering Operations Planning

This topic introduces some of the generally accepted techniques used in software engineering operations planning. Operations engineers must deal with a number of key issues to ensure software operates effectively. Operations engineers should document their software engineering operations steps and tools, using any type, form or medium suitable for the purpose (e.g., Wikis, documents, and more). The following topics are typically considered suitable as evidence of well documented operations:

- Policies and plans,
- Service documentation,
- Procedures,
- Processes, and
- Process control records.

2.1. *Operations Plan and Supplier Management* [1, c4s4.1][3, c6s6.1]

Software engineering operations planning should comprise part of the process of translating project requirements and the needs of the developers and maintainers into services, and it should provide a road map for directing progress. This process often involves the products and services of suppliers that must be well coordinated to ensure quality service. IEEE 20000 describes planning activities, as well as IEEE 12207, which lists the activities operations engineers consider from human, technical and system perspectives.

2.1.1. Operations Plan [1, c4s4.1][3,c6s6.4.12.3a]

Whereas software development typically lasts from some months to a few years, the operations phase usually lasts many years. Therefore, estimating resources is a key element of operations planning. Software engineering operations planning should begin with the decision to develop a new software product and should consider its maintenance and operations requirements early. A concept document should be developed, followed by an operations and maintenance plan [1,c7s2], and both should address the following:

- Scope of the operations and software maintenance,
- Adaptation of the software engineering operations process and tools,
- Identification of the software engineering operations organization,
- Estimate of software engineering operations and maintenance costs.

The next planning step suggests to develop a software engineering operations plan, or concept of operations (CONOPS). This plan should be prepared during software

development and should specify how users will request software modifications and report problems or issues when the software will be operational. Software engineering operations planning is addressed in IEEE 12207 [3] and IEEE 2675 [4]. The standards provide guidelines for planning, implementing, maintaining, automating and supporting production software. Finally, at the highest planning level, the operations organization must conduct business planning activities (e.g., budgetary, financial and human resources), just as all the other divisions of the organization (refer to the Software Engineering Management KA). IEEE 20000 recommends that the operations plan address issues associated with a number of planning perspectives, including the following:

- The roles and responsibilities for implementing, operating and maintaining the new or changed service,
- Activities to be performed by customers and suppliers,
- Changes to the existing service management framework and services,
- Communication to the relevant parties,
- New or changed contracts and agreements to align with changes in business needs,
- Staffing and recruitment requirements,
- Skills and training requirements (e.g., users, technical support),
- Processes, measures, methods and tools to be used in connection with the new or changed service,
- Capacity management, financial management
- Budgets and timescales,
- Service acceptance criteria, and
- The expected outcomes from operating the new service, expressed in measurable terms.

This plan ensures that an operational strategy is defined, conditions for correct operations are identified and evaluated, the software is tested at scale to operate in its

intended environment, and surveillance is provided to ensure responsiveness and availability of the software by ensuring constant support. At the individual request level (e.g., PR and modification request (MR)) need planning. Once individual requests are received and validated, the release or version planning activity requires that operations engineers perform the following tasks:

- Identify the target availability dates of individual requests,
- Agree on the content of subsequent releases or versions,
- Identify potential conflicts and develop alternatives,
- Assess the risk of a given release and develop a rollback and data migration plan (see section 3.2) in case problems arise,
- Inform all stakeholders.

2.1.2. Supplier Management

[1, c7s3][3, c6s6.1]

Supplier management ensures that the organization's suppliers and their performance are managed appropriately to support the seamless provision of quality products and services. IEEE 12207 lists the activities that the operations engineer will perform to establish an agreement to acquire suppliers' products and/or services. From an operations engineer's perspective, the nature of the relationship with suppliers and the approach should be determined by the nature of the products and services needed in a project. Managing suppliers of services related to operational software includes managing outsourced services and services provided by software as a service (SaaS), PaaS and IaaS.

2.2. *Development and Operational Environments*

[2*, c9]

The overall software process requires the use of different environments at different stages. These are typically defined as the development environment, the testing or quality assurance (QA) environment, the preproduction environment, and the production environment. To build quality into the product and reduce the risks associated with the release of software in the production environment (whether the release is associated with new functionality or a defect fix), engineers must ensure that the different environments are all coherent and synchronized with the production environment.

For this reason, DevOps recommends that the creation of all the different environments be automated and built from a single code repository. In mature DevOps organizations, the creation of the different environments is completely automated and made available as a service. Also, all environments need to be built from the same code source (single source of truth) to ensure that all the environments are synchronized with the production environment in which the software is released. This leads to the concept of IaC.

2.3. *Software Availability, Continuity and Service Levels*

[1, c6s6.3]

Service availability and continuity must be managed to ensure that customer commitments are met. Because service availability and continuity are defined as nonfunctional requirements early in a project (see the Software Quality KA), operations engineers will ensure that the proper infrastructure is planned, designed, implemented and tested. Software availability is measured and recorded, and unplanned nonavailability is investigated and appropriate actions taken. Service reports produce availability and continuity indicators of operations services against service-level targets.

The service-level management process monitors the agreed software level of service, including workload characteristics, performance and availability trend information and customer satisfaction analysis. Defining, agreeing to and documenting service-level agreements (SLAs) can help clarify the full range of operations services obligations provided.

2.4. *Software Capacity Management*

[1, c6s6.5]

IEEE 20000 describes the need to ensure that the software product has the capacity, at all times, to meet current and future agreed-upon demands created by the customer's business needs. The current and expected business requirements for services should be understood in terms of what the business needs in order to deliver its products or services to its customers. Business predictions and workload estimates should be translated into specific requirements and documented. The reaction to variations in workload or environment should be predictable; data on current and previous components, as well as resource utilization at an appropriate level, should be captured and analyzed to support the process.

Capacity management is the focal point for all performance and capacity issues. The process should directly support the development of new and changed services by sizing and modeling these services. A capacity plan documenting the actual performance of the infrastructure and the expected requirements should be produced at a suitable frequency (at least annually), considering the rate of change in services and service volumes, information in the change management reports, and changing customer business requirements. The capacity plan should document costed options for meeting business requirements

and recommend solutions to ensure achievement of the agreed-upon service-level targets as defined in the SLA. The technical infrastructure and its current and projected capabilities should be well understood.

2.5. *Software Backup, Disaster Recovery and Failover*

[1, c6s6.3.4]

IEEE 20000 proposes that, to ensure continuity planning and testing, backups of data, documents and software, as well as any equipment and staff necessary for service restoration, should be quickly available following a major service failure or disaster. Backup and data recovery are important activities; successful recovery is especially vital. The need for successful recovery should influence which backup and recovery methods are used (full or incremental), how frequently restore points are established, where they are stored, and how long they are retained.

Preparedness and regular test of backup, disaster recovery, and failover should be constantly rehearsed as changes to the production environment are made. This is another essential activity that is triggered when outage assessments are done. Testing disaster recovery requires stopping the service, identifying the checkpoint state and triggering the failover process. Software engineers should understand that failure is inevitable and that automated failover daemons can reduce recovery time drastically. To achieve this, software applications should include failure-handling logic; this must be planned during development. DevOps can help organizations that want to reduce failovers and disasters by automating and launching tests as often as possible to ensure readiness in case of a failure or catastrophic event.

2.6. *Software and Data Safety, Security, Integrity, Protection and Controls* [1, c6.s6.6]

The need to manage information security effectively within all service activities is described in IEEE 20000. This is done by conducting a software risk assessment on the security and availability of information. Operations engineers should strive to enforce the following controls:

- a) Senior management should define their information security policy, communicate it to staff and customers, and act to ensure its effective implementation,
- b) Information security management roles and responsibilities should be defined and allocated to post holders,
- c) A representative of the management team should be assigned the role of monitoring and maintaining the effectiveness of the information security policy,
- d) Staff with significant security roles should receive information security training,
- e) All staff should be made aware of the information security policy,
- f) Expert help on risk assessment and control implementation should be available,
- g) Changes should not compromise the effective operation of controls, and
- h) Information security incidents should be reported following incident management procedures, and a response should be initiated.

3. Software Engineering Operations Delivery

This topic introduces some of the generally accepted processes used during software engineering operations delivery (IEEE 20000): SLA, service reporting, service continuity, availability management, budgeting and accounting for IT services, capacity management, and information security management.

3.1. *Operational Testing, Verification and Acceptance* [2*,c10] [3, c6s6.3.5.3d]

Software engineers plan and execute software verification as early as possible, using test-driven development (TDD) and acceptance test-driven development (ATDD) techniques and tools that ensure that operational testing is ongoing during the development of the software, not only at the end of a project. DevOps plays an important role in developing and automating software testing services and integrating different tools to boost software productivity and quality. (See TDD and ATDD in the Software Testing KA.)

3.2. *Deployment/Release Engineering* [2*,c12][3,c6s6.3.5.3d]

A software operations engineer's main responsibility relates to the deployment and release of software to ensure its continued performance. As defined in [2*], "deployment is the installation of a specified version of software to a given environment (e.g., deploying code into an integration test environment or deploying code in production)," whereas "release is when we make a feature (or set of features) available to all our customers or a segment of customers (e.g., we enable the feature to be used by 5% of our customer base)." Release processes include all the activities related to release management. ISO 12207 [3] lists release control activities and explains the need to identify and record release requests, identify the software system elements in a release followed by approval, and track the releases in their specified environments.

DevOps advocates integrating development and operations in the same team to improve software engineering operations efficiency. In traditional software processes, when an application is ready for deployment, it is

transferred from a development team to an operations team that is responsible for deployment, which is mostly done manually. This results in processes that are inefficient from both a time and a quality perspective. To improve the efficiency of the deployment process, DevOps calls for automating the different deployment steps, including packaging the code, generating configuration files, restarting the servers, configuring the servers and databases, installing the software on the different servers, launching the execution of the application, and executing smoke testing.

Different release engineering strategies can be used to reduce the risks associated with software releases. These strategies can be grouped into two main categories: environment-based release strategies and application-based release strategies. Environment-based release strategies use a staging environment to support the release of a new version of an application. In other words, the basic strategy involves deploying the new version of the application to a staging environment. Application-based release strategies are based on the use of toggles (e.g., feature toggles) that make it possible to enable or disable specific sections of the code (e.g., a feature) using configuration parameters.

Deployment and release are supported by automation techniques and tools. The canary release testing technique is a partial and time-limited deployment of a change in a service and an evaluation of that change. This evaluation helps the operations engineer decide whether to proceed with a complete deployment. Similarly, tools that manage the installation of new software typically observe the newly started server for a while, ensuring that the server doesn't crash or otherwise misbehave. The same technique is useful for observing recent changes; if they do not pass the validation period, they can be automatically rolled back. The Software Configuration

Management KA provides more information about the release processes. Once the application platform is deployed in the targeted production environment, the decision to make it available to the users (release it) becomes a business decision.

3.3. *Rollback and Data Migration* [2*, c12][3, c6s6.4.10.3]

Rollback and *data migration* are terms used to describe the process of returning software and its database to a state where they work properly. Software engineers ensure that when a new version of the software and its databases have been modified and deployed to production, they can easily and quickly be rolled back in case the new version is causing defects or product degradation in production. This means a planned and rehearsed rollback is done before a new version of the software is deployed in production. DevOps processes automate this process to make it faster; in fact, the automated surveillance can trigger rollback and data migration to a previous state so quickly that the end user doesn't notice that there was a problem. Both release strategy categories (described in section 3.2) — environment-based release and application-based release — can be used to support rollback.

3.4. *Change Management* [1, c9s9.2]

This operations process ensures that all changes are assessed, approved, implemented and reviewed in a controlled manner. All change requests are recorded and classified (e.g., emergency, urgent, major and minor). This process assesses the risk of a change and the need for a rollback strategy in case of failure. Large systems might require that a change schedule be planned with the product manager and end users.

Whereas in traditional software delivery processes (or software life cycle models), all changes are delivered as part of new software releases (containing multiple changes related to different aspects of the application or system) issued at fixed time intervals (e.g., every three months), DevOps aims to deliver small units of change (a single new functionality or service, or defect fix, rather than a new version of an application containing multiple changes) on demand and independently from each other. For this purpose, software applications (or services) must be architected to enable small, independent software deployments.

3.5. *Problem Management*

[1, c8s8.3]

The objective of this operations process is to minimize disruption to the business through the identification and analysis of the cause of software and system incidents and problems. This approach may require the involvement of a multidisciplinary team, whose software engineers and operations engineers investigate, for example, recurring production problems that might have an underlying cause in software infrastructure and system components. This might require monitoring, logging and profiling the software and its infrastructure behavior.

4. Software Engineering Operations Control

This topic introduces some generally accepted techniques used in software engineering operations control.

4.1. *Incident Management*

[1, c8s8.2]

Incident management is the process of recording, prioritizing and assessing the

business impact, resolution, escalation and closure of software incidents. The modern DevOps approach automates software surveillance using alerts and logs to prevent minor incidents from becoming major incidents.

4.2. *Monitor, Measure, Track and Review*

[2*, c14-15]

Software engineering operations activities monitor capacity, continuity and availability. In a DevOps mindset, hope should not be a strategy; instead, engineers should be informed about system quality and operational health with evidence, such as the following key performance indicators (KPI), which are available to stakeholders in real time:

- Production system's monitoring and product telemetry,
- Actionable verification and validation results before and after release to production,
- End-user activity and resource use,
- Impact analysis results,
- Inter- and intra-related dependencies required for system operation,
- Configuration changes unrelated to approved deployment tasks, and
- Security and resilience performance capability.

4.3. *Operations Support*

[1, c6, c14s5]

IEEE 12207 [3], IEEE 20000 [1] and IEEE 2675 [4] identify the primary software engineering operations activities that support the operations processes — activities that operate the software product in its intended environment — and the primary activities that provide support to the customers of the software products. Operations support activities are initiated at the planning stage of the project and are then executed, which often requires

techniques and tools to proactively monitor the product and services and react quickly to events and incidents. Support activities are often described in SLAs.

4.4. *Operations Service Reporting* [1,c6s6.2]

Service reporting aims to produce agreed-upon, timely, reliable and accurate information for decision-making. Each service report helps demonstrate how an operations service has performed and whether it has met some stated and agreed-upon end-user objective. Typical service reports address performance against service-level targets, as well as security breaches, the volume of transactions and resource use, incidents and failures, trend information, and satisfaction analysis. Operations engineers need to establish automated systems and tools for measurement to do the following:

- Determine whether measures are already available or additional instrumentation for collection, analysis and reporting is needed
- Select or develop a framework and tools to allow coordination of measurement collection for analysis, reporting and control

5. Practical Considerations

This topic introduces practical considerations for software engineering operations.

5.1. *Incident and Problem Prevention* [2*, c7]

The overall operations process needs to be automated as much as possible to prevent incidents and problems, and automated testing needs to be integrated throughout the process. Also, product telemetry should be implemented with proper analytics

techniques to detect problems as early as possible to prevent incidents. For this purpose, data collected at all layers of the product stack (including application layer, operating system layer and infrastructure layer) must be collected and analyzed. Using product telemetry not only allows engineers to detect potential issues but also provides the foundation for identifying the source of the problem.

5.2. *Operational Risk Management* [3, c6s6.4.12.3c4]

Operations engineers must manage a number of risks. IEEE 2675 [4] defines continuous risk management as a continuous process that can be automated to monitor operations constantly for risks that can affect software availability, scalability and security. Operations engineers can take measures to automate the alerts. To decide what events will trigger an alert, they need to talk with product owners and software engineers to establish an agreed-upon level of risk tolerance. Other perspectives are to choose the deployment process that is appropriate for the risk profile of a given service and the risks of exposing private data.

5.3. *Automating Software Engineering Operations* [2*, c8]

Automation has taken an important place in recent years in modern operations. Software engineers achieve the best results when coupling applications and operations automation. Although automation primarily focuses on managing the life cycle of a system or infrastructure (e.g., user account creation, environments and server provisioning, runtime config changes), it can also be useful in other use cases where services can be developed to help software engineers deploy, test and debug during development. Trends in operations

automation aim to reduce complexity, accelerate provisioning of infrastructure, offer operations services scripts to developers, define applications, automate deployment and test workflows.

5.4. *Software Engineering Operations for Small Organizations*

Very small organizations (organizations of up to 25 people) have difficulty applying standards developed by and for large organizations, as their requirements can overwhelm the capabilities of small organizations. This is where the ISO/IEC 29110 series is useful, as it provides standards and guidelines adapted to very small organizations to ensure the quality of their software engineering operations [7]. Software engineers should be aware that operations processes can be adapted to small organizations and that the ISO/IEC 29110 series is available for this purpose.

6. Software Engineering Operations Tools [1, c5s5g][2*, c12]

This topic encompasses tools that are particularly important in software engineering operations for maximizing the efficient use of personnel. Automating development, maintenance and operations-related tasks saves engineering resources and improves quality and turnaround. When implemented appropriately, such automated tasks are generally faster, easier and more reliable than they would be if they were attempted manually by software engineers and operations engineers. DevOps supports such automation for integrating, building, packaging, and deploying reliable and secure systems. It combines development, maintenance, and operations resources and procedures to perform CI, delivery, testing and deployment.

Continuous delivery (CD) is a software engineering practice that uses automated tools to provide frequent releases of new systems (including software) to staging or various test environments. CD continuously assembles the latest code and configuration from the head into release candidates.

Continuous testing is a software testing practice that involves testing the software at every stage of the software development life cycle. Continuous testing aims to evaluate the quality of software at every step of the CD process by testing early and often. Continuous testing involves various stakeholders, such as developers, DevOps personnel, and QA and end-users..

Continuous deployment (aka CD) is an automated process of deploying changes to production by verifying intended features and validations to reduce risk. Jez Humble and David Farley [8] pointed out that “[t]he biggest risk to any software effort is that you end up building something that isn’t useful. The earlier and more frequently you get working software in front of real users, the quicker you get feedback to find out how valuable it really is.”

6.1. *Containers and Virtualization*

Different container/virtualization technologies and management tools (also called orchestrators) are available to operations engineers to improve the scalability of applications and standardize software deployment across multiple computer and server suppliers. [4, c6,s6.4.12] Operations engineers use their knowledge of the size and complexity of each project to identify the best tool for flexibility, security and monitoring.

6.2. Deployment

[2*, c12]

Different technologies and tools can be used to manage software deployments in different environments. [4, c5s5.1] Also, different tools are usually combined to cover the different phases and aspects of software deployment, ranging from the specification of deployment and configuration using descriptor files to the automated deployment and management of production environment resources.

6.3. Automated Test

[2*, c10]

To enable fast and constant feedback to the developers, testing must be automated as much as possible throughout the entire software delivery process, including throughout development and operations. For this purpose, a testing strategy covering the different types of test (unit test, integration test, system test, user acceptance test) must be defined, and tools to support and automate the different testing phases must be selected. The automation of testing is critical to provide continuous feedback to software engineers developing code and thereby to improve software quality.

6.4. Monitoring and Telemetry

[2*, c14-15]

Monitoring and telemetry are key aspects of software engineering operations. They collect data at all layers of the software system (including application, operating system and server) and extract information that can be used to analyze and monitor different aspects of the system to detect issues and follow the evolution of various properties. James Turnbull [9] describes a general monitoring framework architecture used by engineering operations in many technology organizations. Implementing monitoring solutions requires combining different techniques and tools to collect data at different layers. This includes logs at the application level, execution traces at the operating system level and resource use information (like CPU and memory use) at the server level. Then, based on the collected data, different analytics techniques (e.g., statistical analysis and machine learning techniques) can be used to extract relevant information. Finally, dashboards can be used to visualize the extracted information; different dashboards can be developed to display relevant information to different stakeholders.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	ISO 20000-1 [1]	The DevOps Handbook [2*]	IEEE 12207 [3]
1. Software Engineering Operations Fundamentals			
<i>1.1. Definition of Software Engineering Operations</i>	c3s3.3		c6s6.4.12
<i>1.2. Software Engineering Operations Processes</i>		s1	c6 s6.4.12
<i>1.3. Software Installation</i>	c3, c6s2	c3s3.1	
<i>1.4. Scripting and Automating</i>		c9	

<i>1.5. Effective Testing and Troubleshooting</i>		c3	
<i>1.6. Performance, Reliability and Load Balancing</i>	c6s6.2		
2. Software Engineering Operations Planning			
<i>2.1. Operations Plan and Supplier Management</i>	c4s4.1		c6s6.1
<i>2.2. Development and Operational Environments</i>		c9	
<i>2.3. Software Availability, Continuity and Service Levels</i>	c6s6.3		
<i>2.4. Software Capacity Management</i>	c6s6.5		
<i>2.5. Software Backup, Disaster Recovery and Failover</i>	c6s6.3.4		
<i>2.6. Software and Data Safety, Security, Integrity, Protection and Controls</i>	c6s6.6		
3. Software Engineering Operations Delivery			
<i>3.1. Operational Testing, Verification and Acceptance</i>		c10	c6s6.3.5.3d
<i>3.2. Deployment/Release Engineering</i>		c12	
<i>3.3. Rollback and Data Migration</i>		c12	c6s6.4.10.3
<i>3.4. Change Management</i>	c9s9.2		
<i>3.5. Problem Management</i>	c8s8.3		
4. Software Engineering Operations Control			
<i>4.1. Incident Management</i>	c8s8.2		
<i>4.2. Monitor, Measure, Track and Review</i>		c14-15	
<i>4.3. Operations Support</i>	c6, c14s5		
<i>4.4. Operations Service Reporting</i>	c6s6.2		
5. Practical Considerations			
<i>5.1. Incident and Problem Prevention</i>		c7	
<i>5.2. Operational Risk Management</i>			c6s6.4.12.3c4
<i>5.3. Automating Software Engineering Operations</i>		c8	
<i>5.4. Software Engineering Operations for Small Organizations</i>			
6. Software Engineering Operations Tools	c5s5g	c12	

<i>6.1. Containers and Virtualization</i>			
<i>6.2. Deployment</i>		c12	
<i>6.3. Automated Test</i>		c10	
<i>6.4. Monitoring and Telemetry</i>		c14-15	

REFERENCES

- [1] IEEE standard, *ISO/IEC/IEEE 20000-1:2013, Information technology — Service management — Part 1: Service management systems requirements*, ed. IEEE, 2013.
- [2*] G. Kim, J. Humble, J. Debois, J. Willis, N. Forsgren, *The DevOps Handbook: How to create world-class agility, reliability and security in technology organizations*, 2nd ed., IT Revolution Press, 2021.
- [3] IEEE standard, *ISO/IEC/IEEE 12207:2017, Systems and software engineering — Software Life Cycle Processes*, ed. IEEE, 2017.
- [4] IEEE standard, *ISO/IEC/IEEE 2675:2021, IEEE*
- [8] J. Humble, and D. Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.
- [9] Turnbull, James. *The art of monitoring*. James Turnbull, 2014.
- Standard for DevOps: Building Reliable and Secure Systems Including Application Build, Package and Deployment*, ed. IEEE, 2021.
- [5] IEEE/ISO/IEC standard, *IEEE/ISO/IEC 24765:2010, Systems and Software Engineering — Vocabulary*, 1st ed.
- [6] B. Beyer, C. Jones, J. Petoff, N. R. Murphy, *Site Reliability Engineering — How Google Runs Production Systems*, O'Reilly Media, 2016.
- [7] ISO/IEC 29110-5-5:2023, *Systems and software engineering — Lifecycle profiles for Very Small Entities (VSEs)*, Agile/DevOps Software Development and Support Guidelines.

CHAPTER 7

SOFTWARE MAINTENANCE

ABBREVIATIONS

API	Application programming interface
CI	Continuous integration
IEC	The International Electrotechnical Commission
IEEE	The Institute of Electrical and Electronics Engineers
ISO	International Organization for Standardization
KA	Knowledge area
LOC	Lines of code
MR	Modification request
PR	Problem report
SCM	Software configuration management
SEE	Software engineering environment
SLA	Service-level agreement
SLI	Service-level indicators
SLO	Service-level objectives
SQA	Software quality assurance
V&V	Verification and validation
XaaS	Anything as a service

INTRODUCTION

Successful software development efforts result in the delivery of a software product that satisfies user requirements. As those requirements and other factors change, the software product must evolve: Once the software is in operation, defects are uncovered, operating environments change, and new user requirements surface. The maintenance phase of the life cycle begins after a warranty period or after post-implementation support delivery, but maintenance activities occur much earlier.

Software maintenance is an integral part of a software life cycle. However, it has not received the same degree of attention as the other software engineering activities. Historically, software development has had a much higher profile than software maintenance. This is now changing as organizations strive to optimize their software engineering investment by ensuring continuous development, maintenance and operation, progressively eliminating the organizational silos among these areas. The growing acceptance of DevOps practices and tools have drawn further attention to the need to continuously evolve software while ensuring its smooth operation to satisfy users, who are demanding quicker turnaround from software engineers than in the past.

In this *SWEBOk Guide*, software maintenance is defined as the totality of activities required to provide cost-effective support for software in operation. Activities to support software operation and maintenance are performed during the pre delivery stage and during the post delivery stage. Pre delivery activities include planning for post delivery operations, maintainability and determining the logistics support needed for the transition from development to maintenance.. Postdelivery activities include software surveillance, modification, training, and operating or interfacing with a help desk.

The Software Maintenance Knowledge Area (KA) is related to all other aspects of software engineering. Therefore, this KA description is linked to all other software engineering KAs in the *Guide*.

BREAKDOWN OF TOPICS FOR SOFTWARE MAINTENANCE

The breakdown of topics for the Software Maintenance KA is shown in Figure 1.

1. Software Maintenance Fundamentals

This section introduces the concepts and terminology that form a basis for

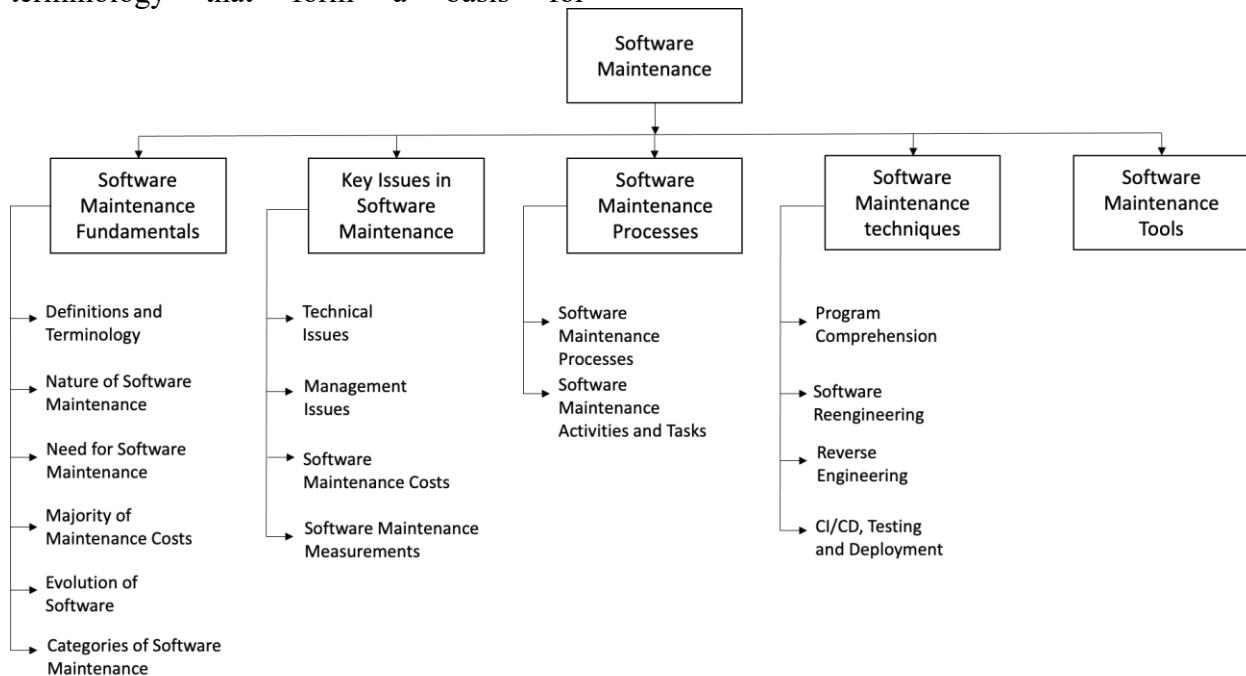


Figure 1. Breakdown of Topics for the Software Maintenance KA

1.1. Definitions and Terminology

[1, s3.1][2*,

c1s1.2, c2s2,2]

The purpose of software maintenance is defined in the international standard for software maintenance: ISO/IEC/IEEE 14764 [1]. In the context of software engineering, software maintenance is essentially one of many technical processes. The objective of software maintenance is to modify existing software while preserving its integrity. The international standard also emphasizes the importance of performing some maintenance activities before final delivery of the software (pre-delivery activities). Software maintenance shares knowledge and tools

understanding the role and scope of software maintenance. Among these concepts are the different categories of software maintenance. Learning about these categories is critical to understanding what this knowledge area encompasses and why it is so important.

with software development and software operation and also has its own processes and techniques

1.2. Nature of Software Maintenance

[2*, c1s1.3]

Software maintenance sustains the software product throughout its life cycle (from development through operations). The software is monitored for capacity, continuity and availability. Modification requests (MRs) and incidents or problems reports (PRs) are logged and tracked, the impact of proposed changes is determined, code and

other software artifacts are modified, testing is conducted, and a new version of the software product is released into operation. Also, training and daily ongoing support are provided to users. A *software maintainer* is defined as a role or an organization that performs software maintenance activities. In this KA, the term sometimes refers to individuals who perform those activities, to contrast their role with the software developer's role.

Maintainers can learn from the developers' and operators' knowledge of the software. Early contact with the developers and early involvement by the maintainers can reduce the overall maintenance costs and efforts. An additional challenge is created when maintainers join the project after the initial developers have left or are no longer available. Maintainers must understand and use software artifacts from development (e.g., code, tests or documentation), support them immediately, and progressively evolve and maintain them over time.

1.3. Need for Software Maintenance

[2*, c1s1.5]

Software maintenance is needed to ensure that the software continues to satisfy user requirements throughout its life span. Maintenance is necessary regardless of the type of software life cycle model used to develop it (e.g., waterfall or Agile). Software products change as a result of both corrective and non-corrective actions. Software maintenance is typically performed to do the following:

- Correct faults and latent defects
- Improve the design or performance of operational software
- Implement enhancements
- Help users understand the software's

functionality

- Adapt to changes in interfaced systems or infrastructure
- Prevent security threats
- Remediate technical obsolescence of system or software elements
- Retire the software

1.4. Majority of Maintenance Costs

[2*,

c4s4.3, c5s5.2]

It is generally accepted that the relative cost of error fixing increases in later phases of the software life cycle. Maintenance also uses a significant portion of the total financial resources attributed throughout the life of a software. A common perception of software maintenance is that it merely fixes faults. However, studies and surveys over the years have indicated that most software maintenance — over 80% — is used for enhancing and adapting the software [3]. Grouping enhancements and corrections together in management reports contributes to a misconception that corrections cost more than they really do. Understanding the categories of software maintenance helps us understand the structure of software maintenance costs — that is, where most of that spending goes [7]. Also, understanding the factors that affect the maintainability of software can help organizations contain costs. Environmental factors that affect software maintenance costs include the following:

- Operating environment (hardware and software)
- Organizational environment (policies, competition, process, product and personnel)

1.5. Evolution of Software

[2*, c3s3.5]

Software maintenance as an activity that supports the evolution of software was first addressed in the late 1960s. Research, by Lehman and others, over a period of twenty years led to the formulation of eight laws of software evolution:

- Continuing Change — Software must be continually adapted, or it becomes progressively less satisfactory.
- Increasing Complexity — As software evolves, its complexity increases unless work is done to maintain or reduce that complexity.
- Self-Regulation — The program evolution process is self regulating with close to normal distribution of measures of product and process attributes.
- Invariant Work Rate — The average effective global activity rate in an evolving software package is invariant over the product's lifetime.
- Conservation of Familiarity — As software evolves, all associated with it (e.g., developers, sales personnel and users) must maintain mastery of its content and behavior to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence, average incremental growth remains invariant as the system evolves.
- Continuing Growth — Functional content of a program must be continually increased to maintain user satisfaction over its lifetime.
- Declining Quality — The quality of software will appear to be declining

unless it is rigorously maintained and adapted to changes in the operational environment.

- Feedback System — Software evolution processes constitute multilevel, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base (in other words, the maintenance process is similar to an Agile process).

Key findings of Lehman's research include a proposal that maintenance is evolutionary development and that maintenance decisions are aided by an understanding of what happens to software over time. Another way to think of maintenance is as continued development that accommodates extra inputs (or constraints) — in other words, large software programs are never complete and continue to evolve. As they evolve, they grow more complex unless action is taken to reduce that complexity.

1.6. Categories of Software Maintenance [1, s3.1.8][2*, c1s1.8, c3s3.3]

Five categories (types) of software maintenance have been standardized to classify a maintenance request: corrective, preventive, adaptive, additive and perfective. ISO/IEC/IEEE 14764 [1], regroups these maintenance categories as either corrections or enhancements, as shown in Figure 2.

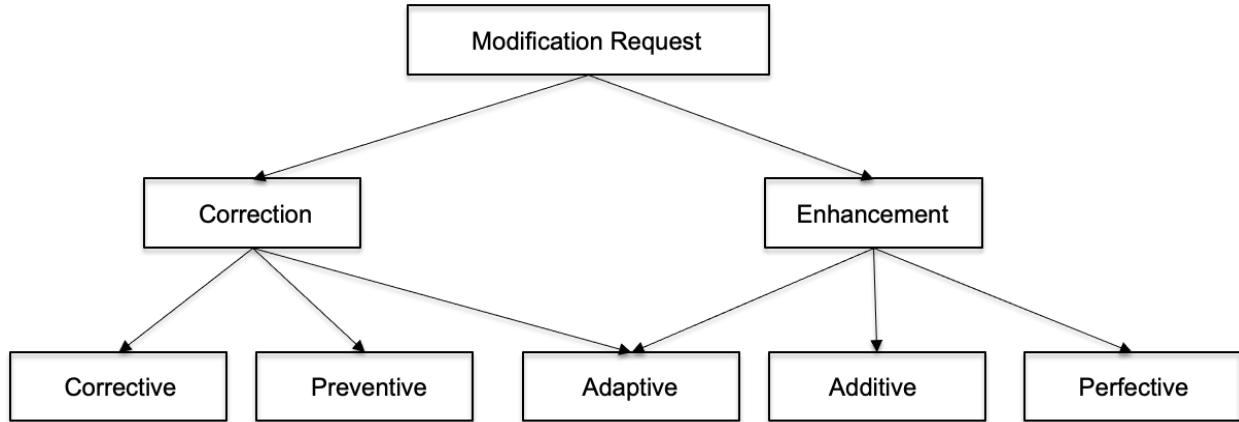


Figure 2. Software Maintenance Categories

ISO/IEC/IEEE 14764 [1] also defines a sixth category — emergency maintenance:

- Corrective maintenance: Reactive modification (or repairs) of a software product performed after delivery to correct discovered problems.
- Preventive maintenance: Modification of a software product after delivery to correct latent faults in the software product before they occur in the live system.
- Adaptive maintenance: Modification of a software product performed after delivery to keep a software product usable in a changed or changing environment. Adaptive maintenance provides enhancements necessary to accommodate changes in the environment in which a software product operates (e.g., an upgrade to the operating system results in changes to the applications).
- Additive maintenance: Modification of a software product performed after delivery to add functionality or features to enhance the usage of the product. Additive maintenance differs from perfective maintenance in that a) it provides additional new functions or features to improve software usability,

performance, maintainability or other software quality attributes, and b) it adds functionality or features with relatively large additions or changes for improving software attributes after delivery..

- Perfective maintenance: Modification of a software product after delivery to provide enhancements for users, improvement of program documentation, and recoding to improve software performance, maintainability, or other software attributes.
- Emergency maintenance: Unscheduled modification performed to temporarily keep a system operational, pending corrective maintenance.

2. Key Issues in Software Maintenance

A number of key issues must be dealt with to ensure the effective maintenance of software. Software maintenance provides unique technical and management challenges for software engineers (e.g.,the challenge of finding a fault in large complex software developed by someone else.)

Similarly, in an Agile setting, maintainers and developers are constantly striving to make sure that clients see the value at the end of each iteration so maintenance activities have to compete with the development of new features for client approval; Planning for a future release, which often includes coding

the next release while sending out emergency patches for the current release, also creates a challenge in balancing maintenance and development work. The following section presents technical and management issues related to software maintenance. They are grouped under the following topics:

- Technical issues
- Management issues
- Maintenance costs
- Maintenance measurement.

2.1. *Technical Issues*

2.1.1. Limited Understanding

[2*, c6s6.9]

Limited understanding describes a software engineer's initial comprehension of software someone else developed. This is reflected in how quickly a software engineer can understand where to change or correct the software. Research indicates that about half of total maintenance effort is devoted to understanding the software to be modified. Naturally, then, the topic of software comprehension is of great interest to software engineers. A number of comprehension factors have been identified: 1) domain knowledge; 2) programming practices (e.g., implementation issues); 3) documentation; and 4) organisation and presentation issues. Comprehension is more difficult in text-oriented representation (e.g., in source code), where it is often difficult to trace the evolution of software through its releases or versions if changes are not documented and the developers are not available to explain them. Thus, software engineers may initially have a limited understanding of the software, and much work must be done to remedy this. Various techniques can help engineers understand existing software, such as visualization and reverse engineering using

tool-based graphical representations of the code.

2.1.2. Testing

[1, s6.2][2*,
c9, c13s13.4.4]

Test planning and activities occur during MRs and PRs processing. The cost of repeating full testing on a major piece of software is significant, in both time and money. To ensure a software modification is validated, the maintainer should replicate or verify changes by planning and executing the appropriate tests — for example, regression testing is important in maintenance. Regression testing is the selective retesting of software or a component to verify that the modifications have not caused unintended effects. Another challenge is finding the time to conduct as much testing as possible. Coordinating tests when different members of the maintenance team are working on different problems at the same time can be challenging. Bringing software offline to test it can be difficult if the software performs critical functions. The Software Testing KA provides additional information and references on software testing and its subtopic on regression testing.

2.1.3. Impact Analysis

[1,
s5.1.6][2*, c13s13.3]

An impact analysis is a complete analysis of the impact of a proposed change in existing software. Software engineers should strive to conduct the analysis as cost-effectively as possible. Maintainers need detailed knowledge of the software's structure and content. They use that knowledge to perform the impact analysis, which identifies all systems and software products that would be affected by a software change request and

develops an estimate of the resources needed to accomplish the change. The analysis also determines the risks involved in making the change. The change request, sometimes called an MR and often called a PR, must first be analyzed and translated into software terms. Impact analysis is performed after a change request enters the software configuration management (SCM) process. ISO/IEC/IEEE 14764 [1] states that the impact analysis tasks do the following:

- Develop an identification scheme for MRs/PRs
- Develop a scheme for categorizing and prioritizing MRs/PRs
- Determine the procedures for an operator to submit an MR/PR
- Identify the information needs and issues that must be tracked and reported to the users and identify the measures that provide feedback on those information needs and issues
- Determine how temporary work-arounds will be provided to the operators
- Track the work-around(s) through to removal
- Determine what follow-up feedback will be provided to the users

Software maintainers often use the severity of a problem as a guide when deciding how and when to fix the problem. The maintainer then identifies the affected components, develops several potential solutions, and, finally, recommends a course of action.

Impact analyses of proposed maintenance changes often consider various factors such as the maintenance action category, the size of the modification, the cost involved, needed to make the modification, and any impacts on performance, safety and security. Designing software with maintainability in mind greatly facilitates impact analysis.

More information can be found in the Software Configuration Management KA.

2.1.4. Maintainability [1, s8.8][2*, c12s12.5.5]

ISO/IEC/IEEE 14764 [1] defines maintainability as the capability of the software product to be modified. Modifications can include corrections, improvements or adaptation of the software to changes in environment, as well as changes in requirements and functional specifications.

As an important software quality characteristic, maintainability should be specified, reviewed and controlled during software development activities in order to reduce maintenance costs. When these activities are carried out successfully, the software's maintainability will benefit. Maintainability is often difficult to achieve because it is often not a primary focus during software development. The developers are typically more focused on other activities and might not pay enough attention to maintainability requirements. This can result in bad architecturing, missing software documentation or test environments, which is a leading cause of difficulties in program comprehension and subsequent impact analysis during maintenance. The presence of systematic and mature software development processes, techniques and tools helps enhance the maintainability of software.

Compromised software maintainability typically increases the burden on software engineers who maintain the software in the future; in other words, it creates technical debt. Technical debt commonly occurs when software engineers attempt to conclude development or maintenance tasks before they have been peer reviewed thoroughly.

This practice generally creates a technical debt that will take additional time and effort to address during maintenance. Specifically, software engineers must investigate three areas in depth when addressing technical debt:

1. Code quality versus relevance: Not all technical debt is urgent.
2. Alignment with organizational objectives: The software architecture should reflect the organization's goals.
3. Process loss: Ensure complementary skills of software engineers involved.

2.2. *Management Issues*

2.2.1. Alignment with Organizational Objectives [1, s9.1.8][2*, c2s2.3.1.2, c3s3.4]

This section describes how to optimize software maintenance activities and economics to be aligned with organizational objectives and the priorities of the business, customers and users.

In many organizations, initial software development is project-based, with a defined time scale and budget. The main goal is to deliver a product that meets user needs on time and within budget. In contrast, software maintenance aims to extend the life of software and keep it operational for as long as possible. In addition, it may be driven by the need to meet user demand for software updates and enhancements.

In both cases, the economics of software maintenance is not as visible as those of software development. At the organizational level, it may be seen as an activity that consumes significant resources with no clear, quantifiable benefit for the organization. As a

consequence, adding new features is often given higher priority than other maintenance activities (such as refactoring, security or performance improvement) to meet the goals and objectives of software customers, as well as with constraints such as time and budget. However, such organizational objectives and constraints must be balanced with software maintainability and engineering standards to avoid code decay and technical debt.

Applying product management approaches to the management of software development and maintenance can help organizations:

- Understand the total cost of operational software over its full life cycle
- Compare the costs and benefits of developing new software versus enhancing existing software
- Resolve staffing and skills issues, as the same team can be responsible for maintenance and development
- Focus more on maintainability requirements from the start, as the same team has responsibility for both development and maintenance

2.2.2. Staffing [1*, s6.4.13.3c][2*, c2s2.3.1.5, c10s10.4]

As software maintenance requires detailed knowledge of software, organizations must be aware of the need to attract and retain software maintenance staff. Since doing only maintenance can be perceived as less interesting; software maintainers can feel they are “second-class citizens,” and morale can suffer as a result, leading to poor performance or high staff turnover. Organizations need to design development and maintenance teams and roles carefully and provide professional development opportunities for their staff.

2.2.3. Process

[1*, s6][2*, c5]

The software life cycle process is a set of activities, methods, practices and transformations that people use to develop and maintain software and its associated products. At the process level, software maintenance activities share much in common with software development (e.g., SCM is a crucial activity in both). Maintenance also requires several activities not found in software development. (Refer to section 3.2.)

2.2.4. Supplier Management [1*, s6.1.2, s8.3, s8.8.2]

Supplier management ensures that the organization's suppliers and their performance are managed appropriately to support the seamless provision of quality products and services when maintenance is contracted to suppliers. The nature of the organization's relationship with suppliers and its approach to supplier management should be determined by the nature of these products and services. Contractors can be hired to conduct maintenance tasks and outsourcing or offshoring software maintenance is a major industry. Outsourcing maintenance means substituting internal capability with an external supplier's capability. Approaches to contracting maintenance include the following:

- Single source or partnership: A single supplier provides all services, or an external service integrator manages the organization's relationship with all suppliers.
- Multi-sourcing: Products and services are provided by more than

one independent supplier. These are combined into a single (software-enabled) service. Multi-sourcing in software services is increasingly common, enabled by the growth of "anything as a service" (XaaS), application programming interfaces (APIs), and data sources.

Many organizations outsource entire portfolios of software. Typically, these portfolios include software that is not mission-critical, as organizations do not want to lose control of the software used in their core business. One major challenge for outsourcers is determining the scope of the maintenance services required, the terms of a service-level agreement (SLA), and the contractual details. Outsourcers need to invest in good communication infrastructure and an efficient help desk staffed with people who can communicate effectively with customers and users [3]. Outsourcing requires a significant initial investment and the setup and review of software maintenance processes that require automation.

2.2.5. Organizational Aspects of Maintenance [1, s9.1.8][2*, c10]

Organizational aspects of maintenance include determining which teams will be responsible for software maintenance. When using Agile life cycle models, the developer also conducts maintenance tasks, acting as both developer and maintainer. Other organizations prefer that the team that develops the software does not necessarily maintain it once it is operational. In deciding where the software maintenance function will be located, software engineering organizations must consider each alternative's advantages and disadvantages. There are a number of disadvantages to having the developer also maintain the

software after it has been put into production, such as the risk that new development will be disrupted when the developers need to attend to failures and the potential loss of knowledge when developers leave the organization, since fewer individuals are familiar with the software; this could also lead to lower-quality documentation, as fewer individuals are involved. However, having a separate maintenance function also has its challenges, as many software engineers do not like limiting their work to maintenance and may be more likely to leave for more interesting work. In addition, a handoff process must be put in place between developers and maintainers, which sometimes leads to friction between teams [3].

The introduction of product management processes has encouraged a single-team approach, particularly for developing and maintaining software that needs to respond rapidly to changes in customer and user needs. Because there are many pros and cons to each option, the decision should be made on a case-by-case basis. What is important is that the organization delegates the maintenance tasks to an experienced group or person and keeps quality documentation on maintenance tasks and all changes made to the software, regardless of the organization's structure.

2.3. *Software Maintenance Costs*

Software engineers must understand the different categories of software maintenance described in 1.6. Presenting costs trends by categories of Maintenance can show customers where maintenance effort is spent for each system supported [7]. The data about maintenance effort by category can be also used to accurately estimate the cost of software maintenance. Cost estimation is an

important aspect of planning software maintenance.

2.3.1. Technical Debt Cost Estimation [1, s6.1.7, s8.8.3.6][2*, c12.12.5]

Technical debt generally makes code more expensive to maintain than it has to be. Technical debt represents the effort required to fix problems that remain in the code when an application is released. Several techniques and indicators can help engineers measure technical debt, including, size, complexity and the number of engineering flaws and violations of good architectural design and coding practices in the source code. ISO/IEC/IEEE 14764 provides suggestions for improving maintainability, including: ensuring legibility, pursuing structured code, reducing code complexity, provide accurate code comments, using indentation and white space, eliminating language weaknesses and compiler dependent constructs, facilitate error-tracing, ensure traceability of code to design, conduct inspections and code reviews. A software product needs to evolve, by adding new features and capabilities, and its codebase must remain maintainable, easily understood, and easy to further evolve. A common barrier to addressing technical debt — or, indeed, of implementing any potential enhancement — is the uncertain reward for doing so. That's why it's so important for organizations to determine the following:

- The quality of their current software
- The current cost of their technical debt
- The potential savings from investing in quality enhancement
- The impact of current quality issues on their business

Furthermore, technical debt is only one factor of several contributing to excess unplanned

work; team or process issues may also need to be understood and addressed. Modern tooling can help detect such issues, which means technical debt should not be handled in isolation but through an examination of its root causes.

2.3.2. Maintenance Cost Estimation [1, s6.2.2, s9.1.4, s9.1.9-10][2*, c12s12.5.6]

An estimate of software maintenance costs should be prepared early in the software planning process [1, c6s1.4]. The costs should be a function of the scope of maintenance activities. ISO/IEC/IEEE 14764 [1, c7s2.4] identified various factors that should be included, such as the following:

- Travel to user locations
- Training for maintainers as well as users
- Cost and annual maintenance for the software engineering environment (SEE) and software testing
- Personnel costs (e.g., salaries, benefits)
- Other resource costs, such as consumables
- Software license maintenance costs
- Product changes, program management
- Field service engineers
- Renting facilities for maintenance

Moreover, as the maintenance and development efforts progress, the estimates should be amended. Historical measurement data should be used as inputs to estimate maintenance costs. Additionally, cost estimates are also required during impact analysis of an MR or PR. The cost estimating method (e.g., parametric model, comparison to analog systems, use of empirical and historical data) should be described. Estimates of individual MRs or PRs typically include the estimated effort associated with

executing a change, resource estimates and an estimated timeline for implementing the change.

2.4. Software Maintenance Measurement [1, s6.1.7][2*, c12]

Measurable software maintenance artifacts include maintenance processes, resources and products [2*, c12s12.3.1]. Measures include size, complexity, quality, understandability, maintainability and effort. One useful measure is the amount of effort (in terms of resources) expended for corrective, preventive, adaptive, additive and perfective maintenance.

Complexity and technical debt measures of software can also be obtained using available tools. These measures constitute a good starting point for the measurement of software quality. Maintainers should determine which measures are appropriate for a specific organization based on that organization's needs. Software measurement programs are discussed in the Software Engineering Management KA.

The software quality model described in the Software Quality KA describes software product and process measures specific to software maintenance. Measurable characteristics of maintainability include the following:

- Modularity measures the degree to which a system or software is composed of components that are independent, such that a change to one component has minimal impact on other components.
- Reusability measures how well a component can be reused.
- Analyzability measures the effort or resources the maintainer must expend either to diagnose deficiencies or causes

- of failure or to identify components to be modified.
- Modifiability measures the maintainer's effort associated with implementing a specified modification without introducing defects or degrading existing product quality.
- Testability measures the effort maintainers and users expend to test the modified software.

Other measures that software maintainers use include the following:

- Reliability: The degree to which a system or software performs specific functions under specified conditions for a specified period, including the following characteristics:
 - Maturity: How well a system or software can meet the need for reliability
 - Availability: Whether a system or software is operational and accessible
 - Fault tolerance: How well a system or software operates despite hardware or software faults
 - Recoverability: How well a system or software can recover data during an interruption or failure
- Size of the software (e.g., functional size, LOC)
- Number of maintenance requests, by time period
- Effort per maintenance request
- Software characteristics (e.g., platform, hardware, programming language, frameworks)

Maintenance measures may be collected, analyzed and trended by category to facilitate improvement and to provide insight into where maintenance costs are expended. The degree of software maintenance effort expended for different applications, listed by category, is valuable business information for users and their organizations. It can also enable the organization to make an internal comparison of software maintenance profiles [7].

3. Software Maintenance Processes

In addition to standard software engineering processes and activities described in ISO/IEC/IEEE 14764 [1], a number of activities are unique to maintainers.

3.1. Software Maintenance Processes

[1, s5.2][2*, c5]

Maintenance processes provide needed activities and detailed inputs and outputs to those activities, as described in ISO/IEC/IEEE 14764 [1]. Maintenance is one of the technical life cycle processes presented in ISO/IEC/IEEE 12207 [9]. Figure 3 shows how maintenance processes connect to other software engineering processes, which interact to support operational software. The software maintenance processes includes the following:

- Prepare for maintenance
- Perform maintenance
- Perform logistics support
- Manage results of maintenance and logistics

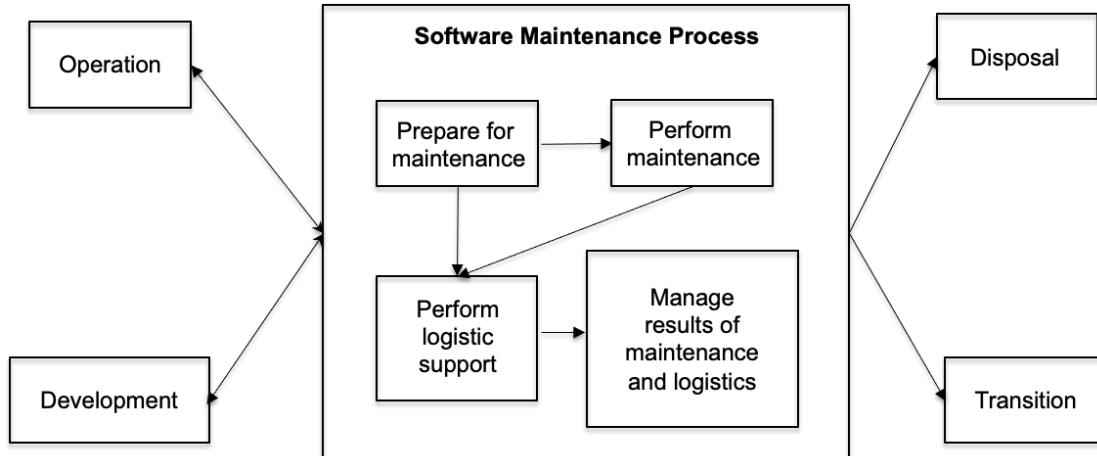


Figure 3. Software Maintenance Processes (ISO/IEC/IEEE 14764) [1]

Recently, Agile methodologies, which promote lightweight processes, have also been adapted to maintenance. This requirement has emerged from the ever-increasing demand for fast turnaround of maintenance services. Improvement to the software maintenance processes is supported by software maintenance maturity models [3].

3.2. Software Maintenance Activities and Tasks [1, s6.1][2*, c6, c7]

The maintenance process contains the activities and tasks necessary to operate and modify an existing software system while preserving its integrity. These activities and tasks are the responsibility of the operator and the maintainer. As already noted, many maintenance activities are similar to those of software development. Maintainers perform analysis, design, coding, testing and documentation. They must track requirements in their activities — just as in development — and update documentation as baselines change. ISO/IEC/IEEE 14764

recommends that when a maintainer uses a development process, the process must be tailored to meet specific needs.

However, there are a number of processes, activities and practices that are specialized to software maintenance:

- Program understanding: This comprises the activities needed to obtain a general knowledge of what a software product does and how the parts work together.
- Transition: This is a controlled and coordinated sequence of activities during which software is transferred progressively from the developer to the operations and maintenance team.
- MR acceptance/rejection: Modifications requesting work greater than the agreed size, level of effort, or level of complexity may be rejected by maintainers and rerouted to a developer.
- Maintenance help desk: The help desk is an end-user and maintenance-coordinated support function that triggers the assessment, prioritization and costing of MRs and incidents.
- Impact analysis: The impact analysis identifies areas impacted by a potential change.
- Maintenance service-level indicators (SLIs), service-level objectives (SLOs), SLAs, and maintenance software and

hardware licenses and contracts: These are contractual agreements that describe the services and quality objectives of third parties.

3.2.1. Supporting and Monitoring Activities [s6.4.13.3d5, s6.1.8][2*, c3s3.4]

Maintainers may also perform ongoing support activities, such as documentation, SCM, verification and validation (V&V), problem resolution, software quality assurance (SQA), reviews, and audits. Another important management of maintenance results activity is that of monitoring customer satisfaction.

3.2.2. Planning Activities [1, s6.1.3, s8.7.2][2*, c10]

An important activity for software maintenance is planning, and this process must address the issues associated with a number of planning perspectives, including the following:

- Business planning (organizational level)
- Maintenance planning (transition level)
- Release/version planning (software level)
- Individual software change request planning (request level)

At the individual request level, planning is carried out during the impact analysis. (See section 2.1.3, Impact Analysis.) The release/version planning activity requires that the maintainer do the following:

- Collect the dates of availability of individual requests
- Agree with users on the content of subsequent releases/versions
- Identify potential conflicts and develop

alternatives

- Assess the risk of a given release and develop a back-out plan in case problems arise
- Inform all stakeholders

Whereas software development projects have a typical duration of months to a few years, the maintenance phase usually lasts many years. Estimating resources is a key element of maintenance planning. Software maintenance planning should begin with the decision to develop a new software product and should consider quality objectives. A concept document should be developed, followed by a maintenance plan, and these should address the following:

- Scope of software maintenance
- Adaptation of the software maintenance processes and tools
- Identification of the software maintenance organization
- Estimate of software maintenance costs

A software maintenance plan should be prepared during software development and should specify how users will request modifications and report problems or issues. Software maintenance planning is addressed in ISO/IEC/IEEE 14764 [1]. Finally, at the highest level of management, the maintenance organization must conduct software maintenance business planning activities (e.g., communications, budgetary, financial and human resources activities). [2*, c10]

3.2.3. Configuration Management [1, s6.1.3c, s6.4.13.3d4][2*, c11s11.3]

ISO/IEC/IEEE 14764 [1] describes SCM as an enabling system or service to support the maintenance process. SCM procedures

should provide for the verification, validation and audit of each step required to identify, authorize, implement and release the software product and its IT assets undergoing change.

It is not sufficient to track MRs or PRs only. Any change made to the software product and its underlying infrastructure must be controlled. This control is established by implementing and enforcing an approved SCM process. The SCM KA discusses SCM in more detail as well as the process by which change requests are submitted, evaluated and approved. SCM for software maintenance differs from SCM for software development in the number of small changes that must be controlled in the operational environment. The SCM process is implemented by developing and following an SCM plan and operating procedures. Maintainers participate in configuration control boards to determine the content of the next release or version in production.

3.2.4. Software Quality [1, s6.1.6, s8.7.2][2*, c13s13.4]

It is not sufficient to simply hope that software maintenance will produce higher quality software. Maintainers should have an effective quality program. They must implement processes to support the continuous improvement of software maintenance processes. The activities and techniques for SQA, V&V, reviews, and audits must be selected in concert with all the other processes to achieve the desired level of quality. It is also recommended that both software operations and maintenance adapt and use the output of the software development process, its techniques and deliverables (e.g., test tools and documentation), and test results. More details

about software quality can be found in the Software Quality KA.

4. Software Maintenance Techniques

This topic introduces generally accepted techniques used in software maintenance.

4.1. Program Comprehension

[2*, c6, c14s14.5]

Programmers spend considerable time reading and understanding programs in order to implement changes. Code browsers are key tools for program comprehension and are used to organize and present source code. Clear and concise documentation also aids program comprehension.

4.2. Software Reengineering

[2*, c7]

Software reengineering refers to the examination and alteration of software to reconstitute it in a new form. It includes the subsequent implementation of the new form. It is often undertaken not to improve maintainability but to replace aging legacy software.

Refactoring is a reengineering technique that aims to reorganize a program without changing its behavior. Refactoring seeks to improve the internal structure and the maintainability of software. Refactoring techniques can be used during maintenance activities to clean up the codebase before and after code changes.

In the context of Agile software development, the incremental nature of continuous integration (CI) often requires the code to be continuously refactored to maintain its quality and reliability. Hence,

continuous refactoring supports the volatile software life cycle by providing better ways to reduce and manage the growing complexity of software systems while improving developer productivity.

4.3. Reverse Engineering

[2*, c7, c14s14.5]

Reverse engineering is the process of analyzing software to identify the software's components and their interrelationships and creating representations of the software in another form or at higher levels of abstraction. Reverse engineering is passive; it does not change the software or result in new software. Reverse engineering efforts typically produce graphical representations of different software artifacts, such as call graphs and control flow graphs from source code. Types of reverse engineering include the following:

- Re-documentation
- Design recovery
- Data reverse engineering — recovering logical schemata from physical databases

Tools are key for reverse engineering and related tasks such as re-documentation and design recovery. Software visualization is a common reverse engineering technique that helps engineers explore, analyze and understand the structure of software systems as well as their development and evolution. Software visualization comprises visually encoding and analyzing software systems, including software development practices, evolution, structure and software runtime behavior using information visualization, computer graphics and human-computer interaction. Generally, software visualization tools are accompanied by various quality assurance features, such as quality metrics

calculation, technical debt estimation, and bad design and coding practices (code smells) detection.

4.4. Continuous Integration, Delivery, Testing and Deployment

[1, s6.4.13.3 Note 1]

Automating development, operation and maintenance-related tasks saves engineering resources. When implemented appropriately, such automated tasks are generally faster, easier and more reliable than they would be if performed manually. ISO14764 that automation includes distribution and installation of software.[1, s6.4.13.3 Note 1]. DevOps supports such automation while building, packaging and deploying reliable and secure systems. It combines development, operations, and maintenance resources and procedures to perform CI, delivery, testing and deployment. [8]

CI is a software engineering practice that continually merges artifacts, including source code updates from all developers on a team, into a shared mainline for building and testing the developed system. With CI, the members of a team can integrate their changes frequently, and each integration can be verified by an automated build (including testing) to detect integration errors as quickly as possible. The fundamental goal of CI is to automatically catch problematic changes as early as possible. CI helps guarantee the working state of a software system at various points from build to release, thereby improving confidence and quality in software products and improving productivity in teams. Specifically, CI automates the build and release processes with continuous build, continuous delivery, continuous testing and continuous deployment. [6, c23, c24].

Continuous delivery is a software engineering practice that enables frequent releases of new systems (including software) to staging or various test environments through the use of automated tools. Continuous delivery continuously assembles the latest code and configuration to create release candidates.

Continuous testing is a software testing practice that involves testing the software at every stage of the software development life cycle. The goal of continuous testing is to evaluate the quality of software at every step of the continuous delivery process by testing early and often. Continuous testing involves various stakeholders such as developers and DevOps, SQA, and operational systems teams.

Continuous deployment is an automated process of deploying changes to production by verifying intended features and validations to reduce risk. As Martin Fowler, in the book *Continuous Delivery*, pointed out, “The biggest risk to any software effort is that you end up building something that isn’t useful. The earlier and more frequently you get working software in front of real users, the quicker you get feedback to find out how valuable it really is.”

5. Software Maintenance Tools

[1,

c6s4][2*, c14]

This topic encompasses tools that are particularly important in software

maintenance where existing software is being modified. Maintenance tools are interrelated with development and operations tools. Together, they are part of the SEE. The following are examples of maintenance tools:

- Configuration management, code versioning and code review tools
- software testing tools,
- Software quality assessment tools (to assess technical debt and code quality)
- Program slicers, which select only the parts of a program affected by a change
- Static analyzers, which allow general viewing and summaries of program content
- Dynamic analyzers, which allow the maintainer to trace the execution path of a program
- Data flow analyzers, which allow the maintainer to track all possible data flows of a program
- Cross-referencers, which generate indexes of program components
- Dependency analyzers, which help maintainers analyze and understand the interrelationships among components of a program

Reverse engineering tools support the process by working backward from an existing product to create artifacts such as specification and design descriptions, which can then be transformed to generate a new product from an old one. Maintainers also use software tests, SCM, software documentation and software measurement tools.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	ISO/IEC/IEEE 14764 2022 [1]	Grubb and Takang 2003 [2*]
1. Software Maintenance Fundamentals		
1.1. Definitions and Terminology	s3.1	c1s1.2, c2s2.2
1.2. Nature of Software Maintenance		c1s1.3
1.3. Need for Software Maintenance		c1s1.5
1.4. Majority of Maintenance Costs		c4s4.3, c5s5.2
1.5. Evolution of Software		c3s3.5
1.6. Categories of Software Maintenance	s3.1.8	c1s1.8, c3s3.3
2. Key Issues in Software Maintenance		
2.1. Technical Issues		
2.1.1. Limited Understanding		c6s6.9
2.1.2. Testing	s6.2	c9, c13s13.4.4
2.1.3. Impact Analysis	s5.1.6	c13s13.3
2.1.4. Maintainability	s8.8	c12s12.5.5
2.2. Management Issues		
2.2.1. Alignment with Organizational Objectives	s9.1.8	c2s2.3.1.2, c3s3.4
2.2.2. Staffing	s6.4.13.3c	c2s2.3.1.5, c10s10.4
2.2.3. Process	s6	c5
2.2.4. Supplier Management	s6.1.2, s8.3, s8.8.2	
2.2.5. Organizational Aspects of Maintenance	s9.1.8	c10
2.3. Maintenance Costs		
2.3.1. Technical Debt Cost Estimation	s6.1.7, s8.8.3.6	c12s12.5
2.3.2. Maintenance Costs Estimation	s6.2.2, s9.1.4, s9.1.9-10	c12s12.5.6
2.4. Software Maintenance Measurement	s6.1.7	c12
3. Software Maintenance Process		
3.1. Software Maintenance Processes	s5.2	c5
3.2. Software Maintenance Activities and Tasks	s6.1	c6, c7
3.2.1. Supporting and Monitoring Activities	s6.4.13.3d5, s6.1.8	c3s3.4
3.2.2. Planning Activities	s6.1.3, s8.7.2	c10

3.2.3. Software Configuration Management	s6.1.3c, s6.4.13.3d4	c11s11.3
3.2.4. Software Quality	s6.1.6, s8.7.2	c13s13.4
4. Software Maintenance Techniques		
<i>4.1. Program Comprehension</i>		c6,c14s14.5
<i>4.2. Software Reengineering</i>		c7
<i>4.3. Reverse Engineering</i>		c7, c14s14.5
<i>4.4. Continuous Integration, Delivery, Testing and Deployment</i>	s6.4.13.3 Note 1	
5. Software Maintenance Tools	s4	c14

FURTHER READING

A. April and A. Abran, *Software Maintenance Management: Evaluation and Continuous Improvement* [3].

This book explores the domain of continuous software maintenance processes. It provides road maps for improving software maintenance processes in organizations. It describes software maintenance practices organized by maturity levels, which allow for benchmarking and continuous improvement. Goals for each key practice area are provided, and the process model presented is fully aligned with the architecture and framework of international standards ISO12207, ISO14764 and ISO15504, as well as models such as ITIL and CoBIT.

IEEE std 2675-2021, IEEE Standard for DevOps: *Building Reliable and Secure Systems Including Application Build, Package and Deployment* [5].

Technical principles and processes to build, package, and deploy systems and applications in a reliable and secure way are specified. Establishing effective compliance and information technology (IT) controls is the focus. DevOps principles presented include mission first, customer focus, left-shift, continuous everything, and systems thinking. How stakeholders, including developers and operations staff, can collaborate and communicate effectively is described. The process outcomes and activities herein are aligned with the process model specified in ISO/IEC/IEEE 12207:2017 and ISO/IEC/IEEE 15288:2015.

REFERENCES

- [1] IEEE standard, *ISO/IEC/IEEE 14764 IEEE Std. 14764:2022, Software Engineering — Software Life Cycle Processes — Maintenance*, third ed: 2022 01, 39p.
- [2*] Grubb, P., and Takang, A. A., *Software Maintenance: Concepts and Practice*, 2nd ed. River Edge, NJ: World Scientific Publishing, 2003.
- [3] April, A., and Abran, A., *Software Maintenance Management: Evaluation and Continuous Improvement*. Wiley-IEEE Computer Society Press, 2008.
- [4] Seybold, C., and Keller, R., Aligning Software Maintenance to the Offshore Reality, 12th European Conference on Software Maintenance and Reengineering. April 1-4, 2008, Athens, Greece, DOI:10.1109/CSMR.2008.4493298.
- [5] IEEE standard, *IEEE Std. 2675-2021, IEEE Standard for DevOps: Building Reliable and Secure Systems Including Application Build, Package and Deployment*, ed: IEEE. 2021.
- [6] Winters, Titus, Tom Mansreck, and Hyrum Wright. *Software engineering at Google: Lessons learned from programming over time*. O'Reilly Media, 2020.
- [7] Abran, A., and Nguyenkim, H., Measurement of the maintenance process from a demand-based perspective, *Journal of Software Maintenance: Research and Practice*, Vol. 5 Issue 2: 63-90, 1993.
- [8] Humble, J. and Fairley, D *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley, 2010
- [9] ISO/IEC/IEEE 12207:2017 *Systems and software engineering — Software life cycle processes*, 2017

CHAPTER 8

SOFTWARE CONFIGURATION MANAGEMENT

ABBREVIATIONS

CCB	Configuration control board
CI	Configuration item
CM	Configuration management
FCA	Functional configuration audit
PCA	Physical configuration audit
QA	Quality assurance
SCCB	Software configuration control board
SCI	Software configuration item
SCM	Software configuration management
SCMP	Software configuration management plan
SCR	Software change request
SCSA	Software configuration status accounting
CMMI	Software Engineering Institute's Capability Maturity Model Integration
SLCP	Software life cycle process
SQA	Software quality assurance
V&V	Verification and validation
KA	Knowledge area
MBX	Model Based Experience
SBOM	Software Bill of Materials
CR	Change Request
VDD	Version Description Document
CMDB	Configuration Management Database

INTRODUCTION

Software configuration management (SCM) is formally defined as “the process of applying configuration management [CM]

throughout the software life cycle to ensure the completeness and correctness of CIs [configuration items],” with CM defined as “a discipline applying technical and administrative direction and surveillance to identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements” [1]. SCM is a software life cycle process (SLCP) that supports project management, development and maintenance activities, quality assurance (QA) activities, and the customers and users of the end product.

The concepts of CM apply to all items controlled, although some differences exist between implementing hardware CM and implementing software CM. CM applies equally to iterative and incremental software development methodology.

SCM is closely related to software quality assurance (SQA). As defined in the Software Quality Knowledge Area (KA), SQA processes ensure that the software products and processes in the project life cycle conform to their specified requirements by requiring software engineers to plan, enact and perform a set of activities that demonstrate that those specifications are built into the software. SCM activities support these SQA goals through software configuration activities (presented later in this chapter). The configuration audit activity can be described as a review of CIs and is closely related to the reviews defined in the quality plan.

The SCM activities should operationalize SCM process management and planning, software configuration identification, software configuration change control, software configuration status accounting (SCSA), software configuration auditing, and software release management and delivery. This operationalization:

1. Determines what is expected to be under control during project development
2. Identifies and records who developed what CI as well as when and where it is allocated
3. Allows controlled changes
4. Tracks CIs' relationships to show how changes that affect one CI might affect other CIs
5. Keeps CI versions under control
6. Ensures that the quality of the CIs delivered meets the requirements for intended use

The SCM KA is related to all other KAs because SCM's object is the artifact produced and used throughout the software engineering process.

BREAKDOWN OF TOPICS FOR SOFTWARE CONFIGURATION MANAGEMENT

Figure 1 shows the breakdown of topics for the SCM KA.

1. Management of the SCM Process [2*, c6, c7]

SCM controls the evolution and integrity of a product by identifying its elements (known as CIs); managing and controlling change; and verifying, recording and reporting on configuration information. From the software engineer's perspective, SCM facilitates development and change implementation activities. Successful SCM implementation requires careful planning and management, which requires a strong understanding of the

organizational context for, and the constraints placed on, the design and implementation of the SCM process. The SCM plan can be developed once for the organization and then adjusted as needed for individual projects.

1.1 Organizational Context for SCM

[2*, c6, ann. D] [3*, Introduction] [4*, c25]

To plan an SCM process for a project, it is necessary to understand the organizational context and the relationships among organizational elements. SCM interacts not just with the particular project but also with several other areas of the organization.

The organizational elements responsible for software engineering supporting processes might be structured in various ways. The overall responsibility for SCM often rests with a distinct part of the organization or with a designated individual. However, responsibility for certain SCM tasks might be assigned to other parts of the organization (such as the development division).

Software is frequently developed as part of a larger system containing hardware and firmware elements. In this case, SCM activities take place in parallel with hardware and firmware CM activities and must be consistent with system-level CM. Note that firmware contains hardware and software; therefore, both hardware and software CM concepts apply.

SCM might interface with an organization's QA activity on issues such as records management and nonconforming items. Regarding the former, project records subject to provisions of the organization's QA program might also be under SCM control. The QA team is usually responsible for managing nonconforming items. However, SCM might assist with tracking and reporting

on software configuration items (SCIs) in this category.

Perhaps the closest relationship is with the software development and maintenance

organizations. It is within this context that many of the software configuration control tasks are conducted. Frequently, the same tools support development, maintenance, and SCM purposes.

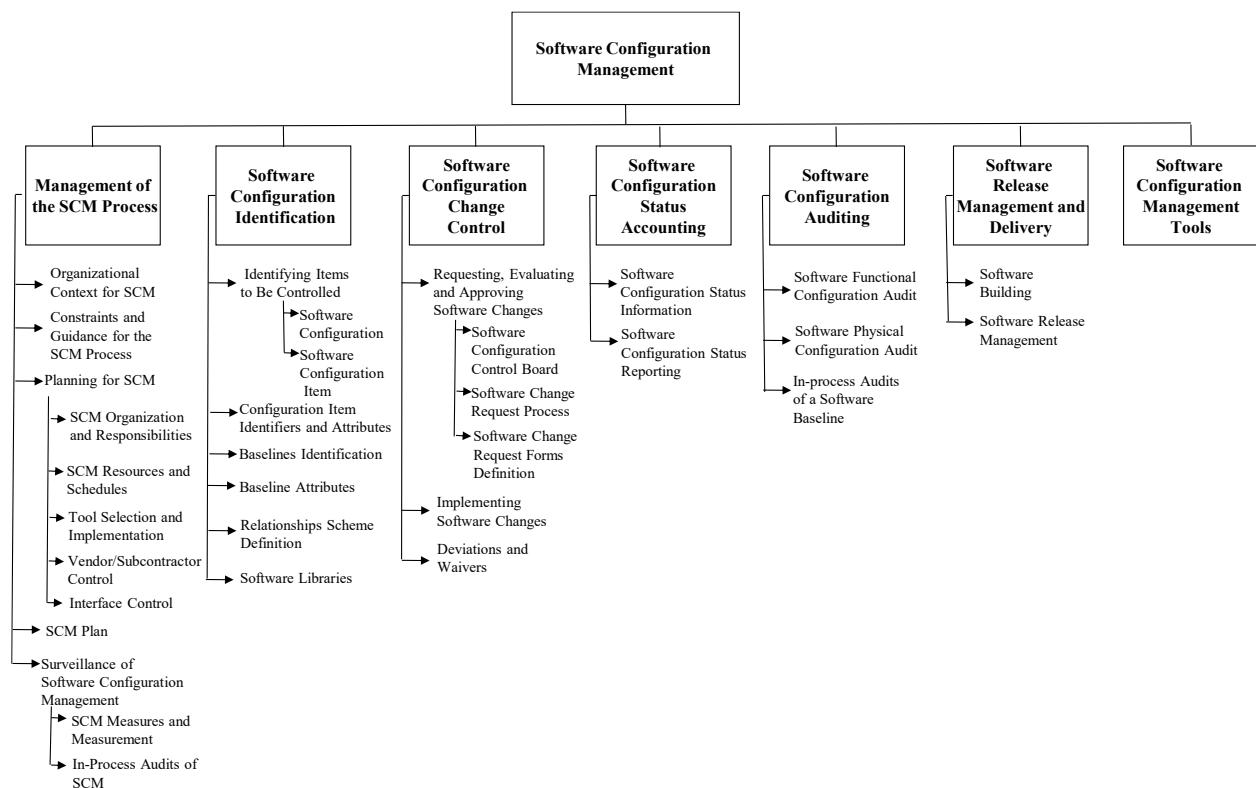


Figure 1. Breakdown of Topics for the Software Configuration Management KA

1.2 Constraints and Guidance for the SCM Process

[2*, c6, ann. D, ann. E] [3*, c2, c5] [5, c19s2.2]

Constraints affecting, and guidance for, the SCM process come from many sources. Policies and procedures set forth at corporate or other organizational levels might influence or prescribe the design and implementation of the SCM process for a project. In addition, the contract between the acquirer and the supplier might contain provisions affecting the SCM process (e.g., certain configuration audits might be required, or the contract

might specify that certain items be placed under CM). When the software to be developed could affect public safety, external regulatory bodies may impose constraints. Finally, the SLCP chosen for a software project and the level of formalism selected for implementation will also affect SLCP design and implementation.

Software engineers can also find guidance for designing and implementing an SCM process in “best practice,” as reflected in the software engineering standards issued by the various

standards organizations. (See Appendix B for more information about these standards.)

1.3 Planning for SCM

[2*, c6, ann. D, ann. E] [3*, c23] [4*, c25]

SCM process planning for a project should be consistent with the organizational context, applicable constraints, commonly accepted guidance and the nature of the project (e.g., size, safety criticality and security). The major activities covered in the plan are software configuration identification, software configuration control, SCSA, software configuration auditing, and software release management and delivery. In addition, issues such as organization and responsibilities, resources and schedules, tool selection and implementation, vendor and subcontractor control, and interface control are typically considered. The planning activity's results are recorded in an SCM plan (SCMP), which is subject to SQA review and audit.

Branching and merging strategies should be carefully planned and communicated because they affect many SCM activities. SCM defines a branch as a set of evolving source file versions [1]. Merging consists of combining different changes to the same file [1]. This typically occurs when more than one person changes a CI. There are many branching and merging strategies in common use. (See the Further Reading section for additional discussion.)

The software development life cycle model chosen (see Software Life Cycle Models in the Software Engineering Process KA) also affects SCM activities, and SCM planning should consider this. For instance, many software development approaches use continuous integration, which is characterized by frequent build-test-deploy

cycles. Clearly, SCM activities must be planned accordingly.

1.3.1 SCM Organization and Responsibilities

[2*, ann. Ds5-6] [3*, c10-11] [4*, c25]

Organizational roles must be clearly identified to prevent confusion about who will perform specific SCM activities or tasks. These responsibilities must also be assigned to organizational entities; this can be made clear by the responsible individual's title or by designating the organizational division or section in addition to the individual responsible within that section. The overall authority and reporting channels for SCM should also be identified, although this might be accomplished at the project management or the QA planning stage.

1.3.2 SCM Resources and Schedules

[2*, ann. Ds8] [3*, c23]

Planning for SCM identifies the resources — including staff and tools — involved in carrying out SCM activities and tasks. It also identifies the necessary sequences of SCM tasks and establishes each task's place in the project schedule and its position relative to milestones established at the project management planning stage. Any training requirements for implementing the plans and training new staff members are also specified.

1.3.3 Tool Selection and Implementation

[3*, c26s2, c26s6]

As in any area of software engineering, the selection and implementation of SCM tools should be carefully planned. The following questions should be considered:

- Organization: What motivates tool acquisition from an organizational perspective?

- Tools: Can we use commercial tools, or do we need to develop our own tools specifically for this project?
- Environment: What constraints are imposed by the organization and its technical context?
- Legacy: How will projects use (or not use) the new tools?
- Financing: Who will pay for the tools' acquisition, maintenance, training and customization?
- Scope: How will the new tools be deployed — for instance, through the entire organization or only on specific projects?
- Ownership: Who is responsible for introducing new tools?
- Future: What is the plan for the tools' use in the future?
- Change: How adaptable are the tools?
- Branching and merging: Are the tools' capabilities compatible with planned branching and merging strategies?
- Integration: Do the various SCM tools integrate among themselves? Do they integrate with other tools in use in the organization?
- Migration: Can the repository maintained by the version control tool be ported to another version control tool while maintaining the complete history of the CIs it contains?

SCM requires a set of tools instead of a single tool. Such tool sets are sometimes called *workbenches*. As part of the tool selection planning effort, the team must determine whether the SCM workbench will be *open* (tools from different suppliers will be used in different SCM process activities) or *integrated* (elements of the workbench are designed to work together).

Organization size and the type of projects involved may also affect tool selection. (See SCM Tools, section 7 of this document)

1.3.4 Vendor/Subcontractor Control

[2*, c13] [3*, c13s9-c14s2]

A software project might acquire or use purchased software products, such as compilers or other tools. SCM planning considers whether and how these items will be managed with configuration control (e.g., integrated into the project libraries) and how changes or updates will be evaluated and managed.

Similar considerations apply to subcontracted software. When a project uses subcontracted software, both the SCM requirements to be imposed on the subcontractor's SCM process and the means for monitoring compliance need to be established. The latter includes determining what SCM information must be available for effective compliance monitoring.

1.3.5 Interface Control

[2*, c12] [3*, c23s4]

When a software item interfaces with another software or with a hardware item, a change to either item can affect the other. Planning for the SCM process considers how the interfacing items will be identified and how changes to the items will be managed and communicated. The SCM role may be part of a larger, system-level process for interface specification and control involving interface specifications, interface control plans and interface control documents. In this case, SCM planning for interface control takes place within the context of the system-level process.

1.4 SCM Plan

[2*, ann. D] [3*, c23]

The results of SCM planning for a given project are recorded in an SCMP, a “living document” that serves as a reference for the SCM process. The SCMP is maintained (updated and approved) as necessary during the software life cycle. For teams to implement an SCMP, they’ll typically need to develop a number of more detailed, subordinate procedures to define how specific requirements will be met during day-to-day activities (e.g., which branching strategies will be used, how frequently builds will occur, how often automated tests of all kinds will be run).

Guidance on creating and maintaining an SCMP, based on the information produced by the planning activity, is available from a number of sources, such as [2*]. This reference provides requirements for information to be contained in an SCMP. An SCMP should include the following sections:

- Introduction (purpose, scope, terms used)
- SCM Management (organization, responsibilities, authorities, applicable policies, directives, procedures)
- SCM Activities (configuration identification, configuration control, etc.)
- SCM Schedules (coordination with other project activities)
- SCM Resources (tools, physical resources and human resources)
- SCMP Maintenance

1.5 Monitoring of Software Configuration Management

[3*, c11s3]

After the SCM process has been implemented, some surveillance may be necessary to ensure that the SCMP provisions are properly carried out. The plan is likely to include specific SQA requirements to ensure compliance with specified SCM processes and procedures. The person responsible for SCM ensures that those with the assigned

responsibility perform the defined SCM tasks correctly. As part of a compliance auditing activity, the SQA authority might also perform this surveillance.

Using integrated SCM tools with process control capability can make the surveillance task easier. Some tools facilitate process compliance while providing flexibility for the software engineer to adapt procedures. Other tools enforce a specific process, leaving the software engineer with less flexibility. Surveillance requirements and the level of flexibility provided to the software engineer are important considerations in tool selection.

1.5.1 SCM Measures and Measurement

[3*, c9s2, c25s2-s3]

SCM measures can be designed to provide specific information on the evolving product, but they can also provide insight into how well the SCM process functions and identify opportunities for process improvement. SCM process measurements enable teams to monitor the effectiveness of SCM activities on an ongoing basis. These measurements are useful in characterizing the current state of the process and providing a basis for comparison over time. Measurement analysis may produce insights that lead to process changes and corresponding updates to the SCMP.

Software libraries and the various SCM tool capabilities enable teams to extract useful information about SCM process characteristics (as well as project and management information). For example, information about the time required to accomplish various types of changes would be useful in evaluating criteria for determining what levels of authority are optimal for authorizing certain changes and in estimating the resources needed to make future changes.

Care must be taken to keep the surveillance focused on the insights that can be gained from the measurements, not on the measurements themselves. Software process and product measurement is further discussed in the Software Engineering Process KA. Software measurement programs are described in the Software Engineering Management KA.

1.5.2 In-Process Audits of SCM

[3*, c1s1]

Audits can be carried out during the software engineering process to investigate the status of specific configuration elements or to assess the SCM process implementation. In-process SCM auditing provides a more formal mechanism for monitoring selected aspects of the process and may be coordinated with the SQA function. (See Software Configuration Auditing.)

2. Software Configuration Identification

[2*, c8]

Software configuration identification identifies items to be controlled, establishes identification schemes for the items and their versions, and establishes the tools and techniques to be used in acquiring and managing controlled items. These activities provide the basis for other SCM activities.

2.1 Identifying Items to Be Controlled

[2*, c8s2.2]

A first step in controlling change is identifying the software items to be controlled. This involves understanding the software configuration within the context of the system configuration, selecting SCIs and developing a strategy for labeling software items.

2.1.1. Software Configuration

Software configuration is the functional and physical characteristics of hardware or software as set forth in technical documentation or achieved in a product. It can be viewed as part of an overall system configuration.

2.1.1 Software Configuration Item

[2*, c8s2.1] [3*, c9]

A CI is an item or aggregation of hardware, software or both, designed to be managed as a single entity. An SCI is a software entity that has been established as a CI [1]. The SCM controls various items in addition to the code itself. Software items with potential to become SCIs include plans, specifications and design documentation, testing materials, software tools, source and executable code, code libraries, data and data dictionaries, and documentation for installation, maintenance, operations and software use.

Selecting SCIs is an important process in which a balance must be achieved between providing adequate visibility for project control purposes and providing a manageable number of controlled items.

2.2 Configuration Item Identifiers and Attributes

[2*, c8s2.3, c8s2.4] [3*, c9]

Status accounting activity (explained later) gathers information about CIs while they are developed. The CIs' scheme is defined in order to establish what information must be gathered and tracked for each CI. Unique identifiers and versions are tracked.

An example scheme may include the following:

CI name
CI unique identifier

CI description
CI date(s)
CI type
CI owner
CI version

The CI's unique Identifier can use significant or nonsignificant codification. An example of significant codification could be XX-YY, where XX is the iteration abbreviation (in case of using an iterative development method) and YY is the CI abbreviation.

2.3 Baseline Identification

[2*, c8s2.5.4, c8s2.5.5, c8s2.5.6]

A software baseline is a formally approved version of a CI (regardless of media type) that is formally designated and fixed at a specific time during the CI's life cycle. The term also refers to a particular version of an agreed-upon SCI. The baseline can be changed only through formal change control procedures. A baseline, with all approved changes to the baseline, represents the current approved configuration. A baseline consists of one or more related CIs.

2.4 Baseline Attributes

[2*, c8s2.5.4]

Baseline attributes are used in the status accounting activity and specify information about the baseline established.

Example baseline attributes may include the following:

Baseline name
Baseline unique identifier
Baseline description
Baseline date of creation
Baseline CIs

2.5 Relationships Scheme Definition

[3*, c7s4]

Relationships provide the connections required to create and sustain structure. The ability to communicate intent and manage the results are significantly enhanced when effective relationships (structuring) are in place (e.g., model-based experience (MBX) platforms). Relationship information exchange and interoperability are needed to support the applicable relationship types. The status accounting activity is responsible for gathering information about relationships among CIs.

Common types of relationships can be described according to the following schemes:

Dependencies: CI-1 and CI-2 depend mutually on each other.

Example: CI-1 depends on CI-2, and vice versa, for instance a class model depends on a sequence diagram, because any change on any of both types of models, affect the other.

CI-1 Code	CI-2 Code	Date
-----------	-----------	------

Derivation: One CI derives from another, typically in a sequential relationship, not because of a lack of resources to handle both CIs but because of a constraint that requires that, for instance, CI-1 is completed before CI-2 is developed.

Example: CI-1 derives from CI-2.

CI-1 Code	CI-2 Code	Date
-----------	-----------	------

Succession: Software items evolve as a software project proceeds. A software item version is an identified instance of an item. It can be thought of as a state of an evolving item. This is what the succession relationship reflects, and it is reflexive in that each CI has this relationship with itself. The first

succession comes up the first time a CI is created. Each time it is changed, a new succession comes up, and tracking these successions is the way to track CI versions.

Example: CI versions along a timeline.

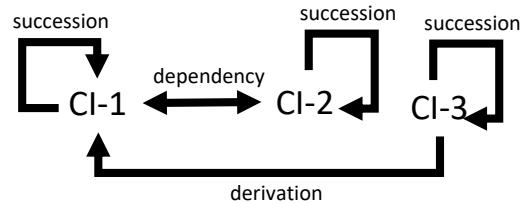
CI Code	Previous Version	Next Version	Date
CI-1	-	1	10/01/2021

Variants are program versions resulting from engineered alternative options. This type of relationship is not as common as the type of relationships described above because it is more expensive to maintain.

The decision on what relationships to track throughout the project is important because tracking some relationships can require extra work. On the other hand, tracking such relationships can facilitate decisions on change requests (CRs) for a CI.

Relationships between CIs can be tracked in a Software Bill of Materials (SBOM). An SBOM is a formal record containing the details and supply chain relationships of the CIs used in building software. CIs in an SBOM are frequently referred to as *components*. Components can be source code, libraries, modules and other artifacts; they can be open source or proprietary, free or paid; and the data can be widely available or access-restricted.

A simple example of the relationships among three CIs in an SBOM, called CI-1, CI-2 and CI-3, is illustrated in Figure 2:



Successions records: According to the scheme defined above, the next table gives each CI was created (three first rows), and the fourth row indicates a change in CI-1 on 2021/10/05.

CI-1	-	1	10/01/2021
CI-2	-	1	10/04/2021
CI-3	-	1	10/03/2021
CI-1	1	2	10/05/2021

Dependency record: According to the scheme defined above for dependencies, CI-2 depends on CI-1, and this dependency relationship was created the day CI-2 was developed.

CI-1	CI-2	10/04/2021
------	------	------------

Derivation record: According to the scheme defined above for derivation, CI-3 depends on CI-1, and this relationship came up the day CI-3 was created.

CI-3	CI-1	10/03/2021
------	------	------------

Figure 1 Example of reported relationships

2.6 Software Libraries

[2*, c8s2.5] [3*, c1s3]

A software library is a controlled collection of source code, scripts, object code, documentation and related artifacts. Requirements and test cases are stored in a repository and should be linked with the code baselines developed. Source code is stored in a version control system, which provides traceability and security for the baselines developed. Multiple development streams are supported in version control systems linked to the binary objects (e.g., object code) derived during the build process. These binary objects are typically stored in a repository that should contain cryptographic hashes used to perform the physical configuration audit (PCA).

The definitive media library contains the release baseline(s) of the artifacts that can be deployed to the test, stage and production systems. The release management process depends on these software libraries to manage the artifacts deployed. In terms of access control and the backup facilities, security is a key aspect of library management.

3. Software Configuration Change Control

[2*, c9] [3*,c8] [4*, c25s3] [5, c11.s3.3]

Software configuration change control is concerned with changes required to CIs during the software life cycle. It covers the process for determining what changes to make, the authority for approving certain changes, support for implementing those changes, and the concept of formal deviations from project requirements as well as waivers of them. Information derived from these activities is useful in measuring change traffic and breakage, as well as aspects of rework.

Given that change to CIs can follow specific rules depending on the industrial sector, area, company, etc., it is very important to identify those rules in the context of the software project for which the SCM process is being developed and to adhere strictly to those rules. The rest of this section can be useful when no specific rules regarding change control exist in the company or the industrial sector where the software project under development is allocated.

3.1 Requesting, Evaluating and Approving Software Changes

[2*, c9s2.4] [3*, c11s1] [4*, c25s3]

The first step in managing changes to controlled items is determining what changes to make. The software change request (SCR)

process (Figure 3) provides formal procedures for submitting and recording CRs; evaluating the potential cost and impact of a proposed change; and accepting, modifying, deferring or rejecting the proposed change. A CR is a request to expand or reduce the project scope; modify policies, processes, plans or procedures; modify costs or budgets; or revise schedules [1]. Requests for changes to SCIs may be originated by anyone at any point in the software life cycle and may include a suggested solution and requested priority. One source of a CR is the initiation of corrective action in response to problem reports. Regardless of the source, the type of change (e.g., defect or enhancement) is usually recorded on the SCR.

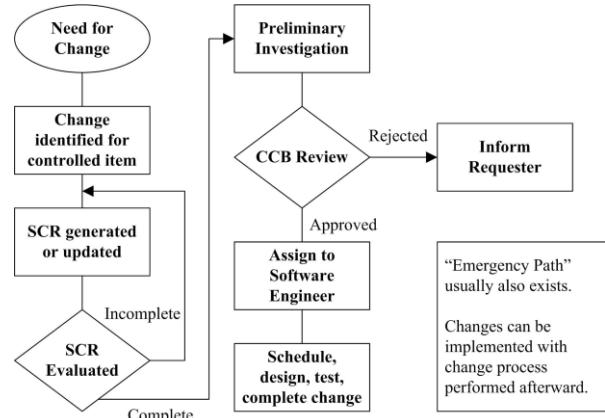


Figure 3. Flow of a Change Control Process

Recording on the SCR enables the software engineers to track defects and collect change activity measurements by change type. Once an SCR is received, a technical evaluation (also known as an *impact analysis*) is performed to determine the extent of the modifications necessary should the CR be accepted. A good understanding of the relationships among software (and, possibly, hardware) items is important for this task. The information recorded about the CIs' relationships could be useful for making decisions affecting any CI, given the potential impact on other CIs. Finally, an established authority — commensurate with

the affected baseline, the SCI involved and the nature of the change — will evaluate the CR's technical and managerial aspects and accept, modify, reject or defer the proposed change.

3.1.1 Software Configuration Control Board

[2*, c9s2.2] [3*, c11s1] [4*, c25s3]

The authority for accepting or rejecting proposed changes rests with an entity known as a configuration control board (CCB). In smaller projects, this authority may reside with the leader or an assigned individual rather than a multi-person board. There can be multiple levels of change authority depending on a variety of criteria — such as the criticality of the item involved, the nature of the change (e.g., impact on budget and schedule), or where the project is in the life cycle. The composition of the CCBs used for a system varies depending on these criteria (but an SCM representative is always present). All stakeholders appropriate to the CCB level are represented. When a CCB's scope of authority is limited to software, the board is known as a Software Configuration Control Board (SCCB). The CCB's activities are subject to software quality audits or reviews.

3.1.2 Software Change Request Process

[3*, c1s4, c8s4] [4*, c25s3]

An effective SCR process requires the use of supporting tools and procedures for originating CRs, enforcing the change process flow, capturing CCB decisions and reporting change process information. Linking this tool capability with the problem-reporting system can facilitate the problem resolution tracking and how quickly solutions are developed.

3.1.3 Software Change Request Forms Definition

[2*, c9s2.3, c9s2.5] [3*, c8s4] [4*, c25s3]

A CR application must include the following:

- A CR form, which must describe the request and give the rationale for it
- A change certification form (necessary if the CR is granted)

These forms can be managed through the corresponding supporting tool, but humans are responsible for designing the forms.

3.2 Implementing Software Changes

[4*, c25s3]

Approved SCRs are implemented using the defined software procedures per the applicable schedule requirements. Because a number of approved SCRs might be implemented simultaneously, a means for tracking which SCRs are incorporated into particular software versions and baselines must be provided. At the end of the change process, completed changes may undergo configuration audits and software quality verification, which includes ensuring that only approved changes have been made. The SCR process typically documents the change's SCM and other approval information.

Changes may be supported by source code version control tools. These tools allow a team of software engineers, or a single software engineer, to track and document changes to the source code. These tools provide a single repository for storing the source code, so they can prevent more than one software engineer from editing the same module at the same time, and they record all changes made to the source code. Software engineers check modules out of the repository, make changes, document the

changes, and then save the edited modules in the repository. If needed, changes can also be discarded, restoring a previous baseline. More powerful tools can support parallel development and geographically distributed environments. These tools may manifest as separate, specialized applications under an independent SCM group's control. They may also appear as an integrated part of the software engineering environment. Finally, they may be as elementary as a rudimentary change control system that is provided with an operating system.

3.3 Deviations and Waivers

[1, c3]

The constraints imposed on a software engineering effort or the specifications produced during the development activities might contain provisions that those working on the project find cannot be satisfied at the designated point in the life cycle. A *deviation* is a written authorization granted before the manufacture of an item to depart from a particular performance or design requirement for a specific number of units or a specific period of time. A *waiver* is a written authorization to allow a CI or other designated item in response to an issue found during production or after the project is submitted for inspection to depart from specified requirements when the CI or project is nevertheless considered suitable for use, either as it is or after rework via an approved method. In these cases, a formal process is used to gain approval for deviations from or waivers of the provisions.

4. Software Configuration Status Accounting

[2*, c10] [3*, c9] [5, c11s3.4]

SCSA is an activity of CM consisting of recording and reporting information needed to manage a configuration effectively regarding CIs, baselines and relationships among CIs. This activity must be done by

following the logical schemes defined in the activity configuration identification for CIs, baselines and relationships for gathering information.

4.1 Software Configuration Status Information

[2*, c10s2.1]

The SCSA activity designs and operates a system for capturing, verifying, validating and reporting necessary information as the life cycle proceeds. As in any information system, the configuration status information to be managed for the evolving configurations must be identified, collected and maintained. In addition, the information itself should be secured where relevant. SCSA information and measurements are needed to support the SCM process and to meet the configuration status reporting needs of management, software engineering, security, performance and other related activities.

The types of information available include but are not limited to the following:

- Ongoing and approved configuration identification
- Current implementation status of changes
- Impacted CIs and related systems
- Deviations and waivers
- Verification and validation (V&V) activities

Automated tools support SCSA as tasks are performed, and reporting is available in a user-friendly format.

4.2 Software Configuration Status Reporting

[2*, c10s2.4] [3*, c1s5, c9s1]

Reported information can be used by various organizational and project elements — including the development team, operations, security, the maintenance team, project

management, software quality activities teams and others. Reporting can take many forms: automated reports, ad hoc queries to answer specific questions, and regular production of predesigned reports, including those developed to meet security, legal or regulatory requirements. In other words, information produced by the SCSA activity throughout the life cycle can be used to satisfy QA and security and to provide evidence of compliance with regulations, governance requirements, etc.

In addition to reporting the configuration's current status, the information obtained by the SCSA can serve as a basis for various measurements. Modern SCM includes a wider scope of information, including but not limited to the following:

- Indicators of integrity (e.g., MAC (Message Authentication Code) SHA1 (Secure Hash Algorithm), MD5 (Message Digest))
- Indicators of security status (e.g., governance risk and compliance)
- Evidence of V&V activities (e.g., requirements completion)
- Baseline status
- The number of CRs per SCI
- The average time needed to implement a CR

5. Software Configuration Auditing

[2*, c11] [5, c11s3.5]

A software audit is an independent examination of a work product or set of work products to assess technical, security, legal and regulatory compliance with specifications, standards, contractual agreements or other criteria [1]. Audits are conducted according to a well-defined process comprising various auditor roles and responsibilities. Because of this complexity, each audit must be carefully planned. An audit can require a number of individuals to

perform various tasks over a fairly short time. Tools to support the planning and conduct of an audit can greatly facilitate the process.

Software configuration auditing determines the extent to which an item satisfies requirements for functional and physical characteristics. Informal audits can be conducted at key points in the life cycle. Two types of formal audits might be required by the governing contract (e.g., a contract covering critical software): the functional configuration audit (FCA) and the PCA. Successful completion of these audits can be a prerequisite for establishing the product baseline.

5.1 Software Functional Configuration Audit

[2*, c11s2.1]

The software FCA ensures that the audited software item is consistent with its governing specifications. The software V&V activities' output (see Verification and Validation in the Software Quality KA) is a key input to this audit.

5.2 Software Physical Configuration Audit

[2*, c11s2.2]

The software PCA ensures that the design and reference documentation are consistent with the as-built software product.

5.3 In-Process Audits of a Software Baseline

[2*, c11s2.3]

Audits can be carried out during the development process to investigate the status of specific configuration elements. In-process audits can be applied to all baseline items to ensure that performance is consistent with specifications or that evolving documentation continues to be consistent with the developing baseline item.

This task applies to every single CI to be approved as part of a baseline. The audit consists of reviewing the CI to determine whether it satisfies requirements. How to conduct the review and the expected result must be described in the quality plan or if there is no quality plan, defined for the software configuration auditing activity.

Continuous reviews of CIs identified in the configuration identification activities help verify conformance to governance and regulatory requirements. Configuration auditing reviews take place throughout project development, whenever a CI must be reviewed.

6. Software Release Management and Delivery

[2*, c14] [3*, c8s2] [4*, c25s4]

In this context, *release* refers to distributing software and related artifacts outside the development activity, including internal releases and distribution to customers. When different versions of a software item are available for delivery (such as versions for different platforms or versions with varying capabilities), re-creating specific versions and packaging the correct materials for version delivery are frequently necessary. The software library is a key element in accomplishing release and delivery tasks.

6.1 Software Building

[4*, c25s2]

Software building constructs the correct versions of SCIs, using the appropriate configuration data, into a software package for delivery to a customer or other recipient such as a team performing testing. For systems with hardware or firmware, the executable program is delivered to the system-building activity. Build instructions help ensure that the proper build steps are taken in the correct sequence. In addition to

building software for new releases, SCM must usually be able to reproduce previous releases for recovery, testing, maintenance or additional release purposes.

Software is built using supporting tools, such as compilers. (See Compiler Basics in the Computing Foundations KA.) For example, if it is necessary to rebuild an exact copy of a previously built SCI, supporting tools and associated build instructions must be under SCM control to ensure the availability of the correct versions of the tools.

Tool capability is useful for selecting the correct versions of software items for a target environment and automating the process of building the software from the selected version and configuration data. This tool capability is necessary for projects with parallel or distributed development environments. Most software engineering environments provide this capability. However, these tools vary in complexity; some require the software engineer to learn a specialized scripting language, while others use a more graphics-oriented approach that hides much of the complexity of an “intelligent” build facility.

The build process and products are often subject to software quality verification. Outputs of the build process might be needed for future reference. They may become records of quality, security, or compliance with organizational or regulatory requirements. The SBOM listing the artifacts included in the build is an important CM output.

In continuous integration, software building is performed automatically when changes to CIs are committed to a source control repository. Tools running on a local or cloud-based server monitor the project’s source

control system and initiate a pipeline of steps to be undertaken every time a change is committed to a particular branch or area of the source code repository. The tool is configured to retrieve a fresh copy of the complete source code for the project and execute the necessary commands to compile and link the code. This configuration is often combined with steps to validate coding standards via automated static analysis, execute unit tests and determine code coverage metrics, or extract documentation from the source code. The resulting artifacts are then deployed through the Release Management process.

6.2 Software Release Management

[4*, c25s2]

Software release management encompasses the identification, packaging and delivery of the elements of a product (e.g., an executable program, documentation, release notes, or configuration data). Given that product changes can occur continually, one concern for release management is determining when to issue a release. The severity of the problems addressed by the release and measurements of the fault densities of prior releases affect this decision. The packaging task identifies which product items are to be delivered and then selects the correct variants of those items, given the product's intended application. The information documenting the physical contents of a release is known as a version description document (VDD). The release notes describe new capabilities, known problems and platform requirements necessary for proper product operation. The package to be released also contains installation or upgrade instructions. The latter can be complicated because some users might have versions that are several releases old. In some cases, release management might be necessary to track the product's distribution to various customers or target systems (e.g., when the supplier was required

to notify a customer of newly reported problems). Finally, a mechanism to help ensure the released item's integrity can be implemented (e.g., by including a digital signature).

A tool capability is needed for supporting these release management functions. For example, a connection with the tool capability supporting the CR process is useful to map release contents to the SCRs that have been received. This tool capability might also maintain information on various target platforms and customer environments.

In continuous delivery, a pipeline is established to build software continuously, as described in the previous section. The resulting artifacts from the build process include executable code and libraries, which can then be combined into an installation package and deployed into an environment for verification or production use.

7. Software Configuration Management Tools

[3*, c26s1]

Many tools can assist with CM at many levels. The scope of these tools varies depending on who uses the tools. CM is most effective when integrated with other processes and by extension with other existing tools. The selection of CM tool can be made depending on the scope that the tool is going to have.

Overview of tools:

- The configuration management system (CMS) provides enabling technology and logic to facilitate CM activities.
- Version control stores the source code, configuration files and related artifacts.

- Build automation (pipeline) is established to enable continuous delivery.
- A repository stores binaries that are created during the build process to extract the latest build artifacts and redeploy them as required — used in the release verification process.
- Configuration management database (CMDB) or similar persistence store.
- Change control tools.
- Release/deployment tools.
- Build handling tools: In their simplest form, such tools compile and link an executable version of the software. More advanced building tools extract the latest version from the version control software, perform quality checks, run regression tests, and produce various forms of reports, among other tasks.
- Change control tools: These tools primarily support the control of CRs and event notifications (e.g., CR status changes, milestones reached).

The CMS supports the unique identification of artifacts. Both individual artifacts and collections are specified in CM systems and related repositories. Structuring creates a logical relationship between artifacts. Validation and release establish the artifacts' integrity, as part of the release management process. Baselines are identified where stability is intended. For example, interface management is identified and controlled, making it part of the baseline process. Change management, including variants and nonconformances, is reviewed and approved, and its implementation is planned. Verification and audit activities are performed as part of the identification, change and release management process. Status and performance accounting are recorded as events occur and are made available through the CMS.

Individual support tools are typically sufficient for small organizations or development groups that do not issue variants of their software products or face other complex SCM requirements. The following are examples of these tools:

- Version control tools: These tools track, document and store individual CIs such as source code and external documentation.

Project-related support tools mainly support workspace management for development teams and integrators. In addition, they can support distributed development environments. Such tools are appropriate for medium-to-large organizations that use variants of their software products and parallel development and do not have certification requirements.

Companywide-process support tools can automate portions of a companywide process, providing support for workflow management, roles and responsibilities. They can handle many items, large volumes of data, and numerous life cycles. In addition, such tools add to project-related support by supporting a more formal development process, including certification requirements.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

Topic	IEEE 828-2012 [2*]	Hass 2003 [3*]	Sommerville 2016 [4*]
1. Management of the SCM Process	c6, c7		
1.1. Organizational Context for SCM	c6, ann.D	Introduction	c25
1.2. Constraints and Guidance for the SCM Process	c6, ann.D, ann.E	c2,c5	
1.3. Planning for SCM	c6, ann.D, ann.E	c23	c25
1.3.1. SCM Organization and Responsibilities	ann.Ds5-6	c10-11	c25
1.3.2. SCM Resources and Schedules	ann.Ds8	c23	
1.3.3. Tool Selection and Implementation		c26s2, s6	
1.3.4. Vendor/Subcontractor Control	c13	c13s9-c14s2	
1.3.5. Interface Control	c12	c23s4	
1.4. SCM Plan	ann.D	c23	
1.5. Surveillance of Software Configuration Management		c11s3	
1.5.1. SCM Measures and Measurement		c9s2; c25s2-s3	
1.5.2. In-Process Audits of SCM		c1s1	
2. Software Configuration Identification	c8		
2.1. Identifying Items to Be Controlled	c8s2.2	c1s2	
2.1.1. Software Configuration			
2.1.2. Software Configuration Item	c8s2.1	c9	
2.2. Configuration Item Identifiers and Attributes	c8s2.3 c8s2.4	c9	
2.3. Baseline Identification	c8s2.5.4 c8s2.5.5 c8s2.5.6		
2.4. Baseline Attributes	c8s2.5.4		
2.5. Relationships Scheme Definition		c7s4	
2.6. Software Libraries	c8s2.5	c1s3	
3. Software Configuration Change Control	c9	c8	c25s3
3.1. Requesting, Evaluating and Approving Software Changes	c9s2.4	c11s1	c25s3
3.1.1. Software Configuration Control Board	c9s2.2	c11s1	c25s3
3.1.2. Software Change Request Process		c1s4, c8s4	c25s3
3.1.3. Software Change Request Forms Definition	c9s2.3 c9s2.5	c8s4	c25s3
3.2. Implementing Software Changes			c25s3
3.3. Deviations and Waivers			
4. Software Configuration Status Accounting	c10	c9	
4.1. Software Configuration Status Information	c10s2.1		
4.2. Software Configuration Status Reporting	c10s2.4	c1s5; c9s1	
5. Software Configuration Auditing	c11		
5.1. Software Functional Configuration Audit	c11s2.1		
5.2. Software Physical Configuration Audit	c11s2.2		
5.3. In-Process Audits of a Software Baseline	c11s2.3		
6. Software Release Management and Delivery	c14	c8s2	c25s4
6.1. Software Building			c25s2
6.2. Software Release Management			c25s2
7. Software Configuration Management Tools		c26s1	

FURTHER READING

S. P. Berczuk and B. Appleton., *Software Configuration Management Patterns: Effective Teamwork, Practical Integration* [6].

This book expresses useful SCM practices and strategies as patterns. The patterns can be implemented using various tools, but they are expressed in a tool-agnostic fashion.

CMMI for Development, Version 1.3, pp. 137-147 [7].

This model presents a collection of best practices to help software development organizations improve their processes. At maturity level 2, it suggests CM activities.

B. Aiello & L. A. Sachs. (2011). Configuration management best practices: Practical methods that work in the real world (1st edition) [8].

This book presents the seven types of change control (Chapter 4, Section 3).

REFERENCES

- [1] IEEE. ISO/IEC/IEEE 24765:2017(E): ISO/IEC/IEEE International Standard — Systems and software engineering — Vocabulary.
- [2*] IEEE. IEEE Standard 828-2012, Standard for Configuration Management in Systems and Software Engineering, . 2012.
- [3*] A. M. J. Hass. *Configuration Management Principles and Practices*, 1st ed. Boston: Addison-Wesley, 2003.
- [4*] I. Sommerville. *Software Engineering*, 10th ed. Global ed. Pearson. 2016.

- [5] J. W. Moore. *The Road Map to Software Engineering: A Standards-Based Guide*, 1st ed. Hoboken, NJ: Wiley-IEEE Computer Society Press, 2006.
- [6] S. P. Berczuk and B. Appleton. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*: Addison-Wesley Professional, 2003.
- [7] *CMMI for Development*, Version 1.3, Software Engineering Institute, 2010.
- [8] B. Aiello & L. A. Sachs. (2011). Configuration management best practices: Practical methods that work in the real world (1st edition).

CHAPTER 9

SOFTWARE ENGINEERING MANAGEMENT

ABBREVIATIONS

<i>PMBOK® Guide</i>	<i>Guide to the Project Management Body of Knowledge</i>
SDLC	Software development life cycle
SEM	Software engineering management
SQA	Software quality assurance
<i>SWX</i>	<i>Software Extension to the PMBOK® Guide</i>
WBS	Work breakdown structure
PSM	Practical Software and Systems Measurement

INTRODUCTION

Software engineering management (SEM) can be defined as a collection of work activities involved with planning, estimating, measuring, controlling, coordinating, leading and managing risk factors for a software project to help ensure that software products and software engineering services are delivered efficiently, effectively and to the stakeholders' benefit [3]. Although project management and measurement management are often seen as separate areas, and each possesses many unique attributes, the close relationship between the two has led to their combined treatment in this knowledge area (KA).

In one sense, it should be possible to manage a software engineering project in the same way other complex endeavors are managed, using models, technical processes and problem-solving styles as other engineering projects do. However, software engineers use different process models, technical processes, and problem-solving styles than other engineers, making these choices based on their education and experience and on the differences between physical and software attributes. Software system elements are logical constructions expressed in logarithmic form, while physical system elements are realized in mechanical, electrical, chemical, biological and other physical media. Software is intangible because it has no physical properties and is malleable because of the relative ease with which code can be modified. Obtaining the desired effect by modifying software code might not be easy, but code modifications, *per se*, are straightforward compared with the modification of physical elements that have already been constructed [12].

As software and software-embedded systems become bigger, more complex, and more intertwined, software management and engineering roles are evolving in response [10], because skilled individuals must actively develop and maintain these systems. Consider the following: hardware is different from software (and not all software is the same). Hardware can be developed, procured, and maintained in a linear fashion. Software is an enduring capability that must be supported and continuously improved throughout its life cycle [13]. Furthermore, the malleable nature of software allows

iteration among and interleaving of the development phases (to a much greater degree than is possible when developing physical artifacts).

Software is made by people and for people, so digital talent matters. Software projects are increasingly important, and their ongoing success largely depends on people with the right skills, knowledge and abilities. This fact is essentially actual and necessary but not sufficient. Other human factors may affect the project's success. During the software development lifecycle, it is impossible to separate the human factors from the technical ones. Therefore, people management activities, such as team and teamwork, leadership, communication, and coordination activities, are important to project success.

Factors such as cultural differences and diverse attitudes may affect the development team. A significant number of software projects failed due to social issues. Poor quality developers may produce poor quality products resulting in a higher cost of reworks to remove their poor-quality traces.

Other issues can complicate effective management of software projects and software life cycle processes, including the following:

- Clients often do not know what is needed or what is feasible.
- Increased understanding and changing conditions will likely generate new or changed software requirements.
- Clients often do not appreciate the complexities inherent in software engineering, particularly regarding the impact of changing requirements.
- As a result of changing requirements and software malleability, software is often

built iteratively rather than as a linear sequence of phases.

- Software is nominally an enduring capability that must be supported and continuously improved throughout its lifecycle
- Software construction differs from hardware implementation in that design is usually part of software construction, whereas in hardware-oriented systems, design precedes hardware implementation to “get it right” prior to procurement or fabrication of hardware [12].
- Software engineering necessarily incorporates creativity and discipline. Maintaining an appropriate balance between the two is sometimes difficult [5].
- The development of software capabilities often involves a high degree of novelty and complexity.
- Typically, the underlying technology has a high rate of change.
- Computer software has become a key component of most modern systems. Software has been elevated to a highly prominent role because of its flexibility and relatively low replication cost compared with hardware.
- Software, as an intangible deliverable, is challenging to measure. Physical measurement units such as the length and weight measures are challenging to apply to the software. This difficulty impacts how to plan, monitor, and control software development projects.
- Software rework to remove faults and respond to change
- Speed and cycle time are important metrics for managing software. Software capabilities are often delivered at increasing speed to satisfy business and mission needs [13].

SEM activities occur on three levels: organizational and infrastructure management, project management, and

management of the measurement program. The last two are covered in detail in this KA description. This fact does not diminish the importance of organizational and infrastructure management issues but rather points out that software organizational engineering managers should be conversant with the project management and software measurement knowledge described in this KA. They should also possess some target domain knowledge. Likewise, it also helps for managers of complex projects and programs where software is part of the system architecture to know what issues software engineering processes (versus other types of engineering processes) introduce into project management and project measurement.

Other aspects of organizational management affect software engineering — for example, organizational policies and procedures that provide the framework for software engineering projects. These policies and procedures might need to be adjusted for effective software development and maintenance requirements. In addition, several policies specific to software engineering might need to be in place or established for the effective management of software engineering at the organizational level. For example, policies are usually necessary to establish specific organization-wide processes or procedures for software engineering tasks such as software design, software construction, estimating, monitoring and reporting. Such policies are important for effective long-term management of software engineering projects across an organization (e.g., one such policy could establish a consistent basis for analyzing past project performance and implementing improvements).

Another important aspect of organizational management is the use of personnel management policies and procedures for hiring, training and mentoring — not only

for a project's success, but also for the organization's long-term success. Given the projected scarcity of skilled software engineers, it is important to provide an environment that attracts and retains good talent. Software engineering personnel can present unique training or personnel management challenges (e.g., maintaining currency in a context where the underlying technology undergoes rapid and continuous change) as part of career development.

Communication management is also often mentioned as an overlooked but important aspect of success in a field where a precise understanding of user needs, software requirements and software designs is necessary. Furthermore, portfolio management is desirable, which provides an overall view of software under development in various projects and programs (integrated projects) of planned software, and of software already in use in an organization. Also, software reuse can be a key factor in maintaining and improving productivity and competitiveness. Effective reuse requires a strategic vision that reflects the advantages and disadvantages of reuse.

Software engineers should have a sound understanding of the aspects of management that are unique to software projects, and they should also have some knowledge of the more general aspects of management discussed in this KA (even in the first few years after graduation).

Certain attributes of organizational culture and behavior, as well as management of functional areas of the enterprise outside the immediate software engineering realm, can influence an organization's software engineering processes, albeit indirectly. Software projects are often targeted at changing the way people work — but culture change is difficult, complicated and unlikely to succeed without a significant effort. For this reason, leadership is an

important attribute for program managers, as they often need to lead the charge for digital transformation. They might need to galvanize their teams and other stakeholders to bring their very best to every project pursuing major change.

Extensive information concerning software project management can be found in the *Guide to the Project Management Body of Knowledge (PMBOK® Guide)* and the *Software Extension to the PMBOK® Guide (SWX)* [1, 2]. Each of these guides includes 10 project management KAs: project integration management, project scope management, project time/schedule management, project cost management, project quality management, project resource/human management, project communications management, project risk management, project procurement management and project stakeholder management. Each KA has direct relevance to this SEM KA.

Additional information is also provided in this KA's references and list of further reading.

This SEM KA discusses the software project management processes shown as the first five topics in Figure 1 (Initiation and Scope Definition, Software Project Planning, Software Project Enactment, Review and Evaluation, Closure), as well as Software Engineering Measurement (the sixth topic shown in the figure) and Software Engineering Management Tools (the seventh topic).

Unfortunately, a common perception of the software industry is that software products often are delivered late, are over budget, are of poor quality and have incomplete functionality. Measurement-informed management — a basic principle of any true engineering discipline (see Measurement in the Engineering Foundations KA) — can help improve perception and reality. In

essence, management without measurement (qualitative and quantitative) suggests a lack of discipline, and measurement without management suggests a lack of purpose or context. To be effective, software engineers must use both measurement and management.

The following working definitions are adopted here:

- *Management* is a system of processes and controls required to achieve the strategic objectives set by the organization.
- *Measurement* refers to the assignment of values and labels to software engineering work products, processes and resources, plus the models derived from them, whether these models are developed using statistical or other techniques [3*, c7, c8].

The software engineering project management sections in this KA use the Software Engineering Measurement section extensively.

This KA is closely related to others in the *SWEBOK Guide*; reading the following KA descriptions will be particularly helpful in understanding this one:

- The Engineering Foundations KA describes some general measurement concepts that directly apply to the Software Engineering Measurement section of this KA. In addition, the concepts and techniques presented in the Statistical Analysis section of the Engineering Foundations KA apply directly to many topics in this KA.
- The Software Requirements KA describes activities that should be performed during the project's Initiation and Scope Definition phase.
- The Software Configuration Management KA deals with the identification, control, status accounting and auditing of software configurations, along with software release

management and delivery and software configuration management tools.

- The Software Engineering Process KA describes software life cycle models and the relationships between processes and work products.
- The Software Quality KA emphasizes quality as a management goal and as an aim of many software engineering activities.
- The Software Engineering Economics KA discusses how to make software-related decisions in a business context.

BREAKDOWN OF TOPICS FOR SOFTWARE ENGINEERING MANAGEMENT

Because most software development life cycle (SDLC) models require similar activities that may be executed in different ways, the topic breakdown, shown in Figure 1, is activity-based. The top-level elements shown in the figure are activities that are usually performed when a software development project is being managed, regardless of which SDLC model is being used (see Software Life Cycle Models in the Software Engineering Process KA). This breakdown does not recommend a specific life cycle model. However, it is important to note the impact the choice of software can have on business success and the associated development of software life cycle models to accommodate changing business needs.

Delivery speed, continuous adaptation and frequent modular upgrades to deliver new capabilities are often key business differentiators and project management imperative [11, 13]. These imperatives should be balanced with risk management activities.

Several software life cycle process models have been and are being developed to shorten development cycles in response to the changing business needs. Most of these

processes involve Agile SDLC approaches [14]. The Agile approach assumes that teams can develop high-quality, adaptive software using continuous design improvement principles and testing based on rapid feedback and change. In comparison, the traditional approach assumes that software-intensive systems are fully specifiable and predictable and can be built through meticulous and extensive planning. The management style associated with the Agile approach emphasizes leadership and collaboration at the team level, whereas the management style of the highly predictive approach is more formal (top-down). Many Agile approaches integrate different management approaches.

For example, Dev/Sec/Ops is a culture and an Agile approach to modern software delivery that aligns development (Dev), security (Sec) and operations (Ops) groups into an integrated team focused on continuous, incremental delivery of capabilities. The main characteristic of Dev/Sec/Ops is that this approach automates, continuously monitors and applies security at all phases of the software life cycle: plan, develop, build, test, release, deliver, deploy, operate and monitor. In Dev/Sec/Ops, testing and security are shifted to the left through automated unit, functional, integration and security testing. This is a key Dev/Sec/Ops differentiator; security/quality assurance (QA) and other nonfunctional and functional capabilities are tested and built simultaneously [11, 14]. Whereas Dev/Sec/Ops encompasses the culture and processes that enable rapid, continual delivery of cyber-resilient systems, complex software-embedded systems can have additional demands that must also be integrated into the Dev/Sec/Ops culture and processes, such as safety. Elevating these demands to be on par with Dev/Sec/Ops highlights the importance

of incorporating quality into all program aspects. The complexity of the end-to-end Dev/Sec/Ops tools and of using emerging technologies such as artificial intelligence (AI) and machine learning (ML) to leverage those tools adds another dimension [15]. For example, Agile and DevOps approaches are reasonably well-established, but in case of AI-based software, new SLDCs maybe required to manage the complexity brought by AI to the software.

It is important to understand the difference between *phases* and *activities* and why an activities breakdown is used. The Project Management Institute (PMI) describes a *phase* this way: “The completion and approval of one or more deliverables characterizes a project phase.” A *deliverable* is a measurable, verifiable work product such as a specification, feasibility study report, detailed design document or working prototype. Some deliverables correspond to part of the project management process, whereas others are the end products or components of the end products for which the project was conceived. The deliverables, and hence the phases, are part of a generally sequential process designed to ensure proper control of the project and to attain the desired product or service, which is the project’s objective. From a project management perspective, phases help accomplish project objectives and maintain control over the project.

The activity-based breakdown in Figure 1 shows what happens but does not imply when, how or how many times each activity occurs. The seven topics are the following:

- Initiation and Scope Definition, which deals with the decision to embark on a software engineering project
- Software Project Planning, which addresses the activities undertaken to prepare for a successful software engineering project from the management perspective
- Software Project Enactment, which deals with generally accepted SEM activities that occur during a software engineering project’s execution
- Review and Evaluation, which deals with ensuring that technical, schedule, cost and quality engineering activities are satisfactory
- Closure, which addresses the activities accomplished to complete a project
- Software Engineering Measurement, which deals with the effective development and implementation of measurement programs in software engineering organizations
- Software Engineering Management Tools, which describes the selection and use of tools for managing a software engineering project

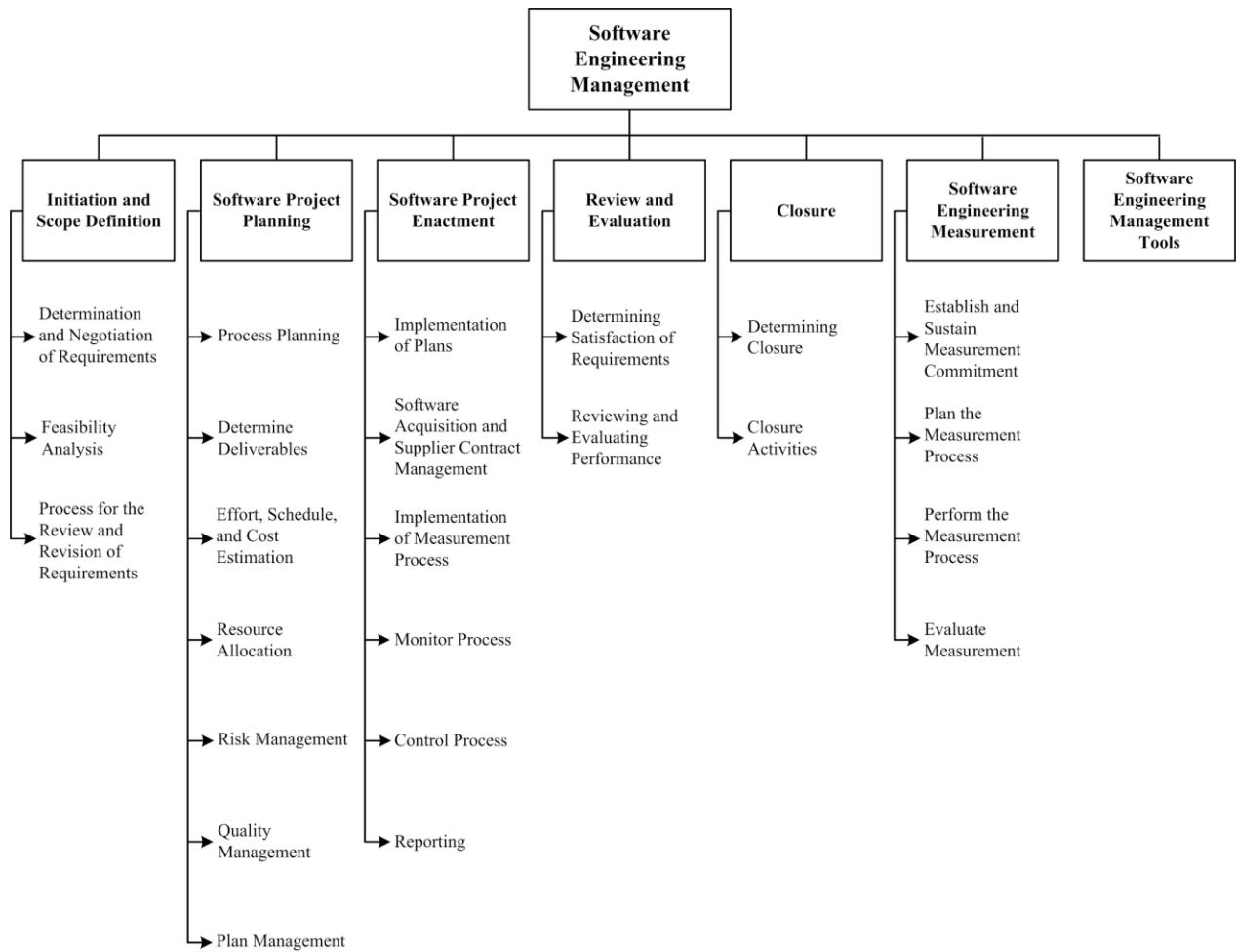


Figure 1. Breakdown of Topics for the Software Engineering Management KA

1. Initiation and Scope Definition

Project initiation focuses on reviewing the software requirements and determining the need, scope, feasibility, and authorization for a software project. Once project feasibility has been established, the remaining tasks in this section are specifying the software requirements and selecting the processes for requirements revision and review.

1.1. Determination and Negotiation of Requirements [3*, c3]

Determining and negotiating the project requirements are the overarching goals of the tasks undertaken during this phase (see the Software Architecture KA and Software Requirements KA). Activities software

requirements review (e.g., elicitation, analysis, specification, and validation). Methods and techniques should be selected and applied considering the various stakeholder perspectives. Requirements provide the basis for all that follows on a software project and are captured in a Project Charter or other high-level project initiation document.

1.2. Feasibility Analysis [4*, c5]

The purpose of the feasibility analysis is to develop a clear description of project objectives and to evaluate alternative approaches to determine whether the proposed project solution is the best approach, given the constraints of technology, resources, finances and changes to ethical, environmental, and socio-

technical considerations. An initial project and product scope statement, project deliverables, project duration constraints, and an estimation of resources needed should be prepared.

Resources (which can be internal or external to the organization) include infrastructure, support, and people with the necessary core competencies. The feasibility analysis often requires estimations of effort and cost based on appropriate methods. (See Section 2.3, Effort, Schedule and Cost Estimation.)

An initial work breakdown structure (WBS) and context diagram may be developed during the project's Initiation and Scope Definition phase activities. Breaking work into smaller tasks is a common productivity technique that makes the work more manageable and approachable. As the project tool that uses this technique, WBS is an important project management document. While a WBS can be used to organize cost and schedule tracking, the WBS does not itself include cost and schedule baselines. Schedules are developed as part of the next activity, project planning (section 2).

An engineering context diagram defines the boundary between the system (or a part of the system) and its environment, showing the entities interacting with it. This document is important in defining management and technical interfaces and trade-offs that must be considered [1]. While engineers are developing the WBS, they should consider all configuration items as tasks to have under control.

1.3. Process for the Review and Revision of Requirements [3, c3]*

Given the inevitability of change, stakeholders should agree on how requirements and scope will be reviewed and revised (e.g., change management and trade-off procedures, iterative cycle retrospectives). (See the Requirements KA.) This indicates that scope and requirements

will not be “set in stone” but can and should be revisited at predetermined points as the project unfolds (for example, at the time when backlog priorities are created or at milestone reviews). If changes are accepted, then forward or backward traceability analysis and risk analysis should be used to ascertain the impact of those changes. For example, backward traceability may link the test script to its associated requirement and design. This link helps monitor the status of requirements satisfaction and helps make decisions to stop testing. It also helps in making tradeoffs regarding requirements and design. (See Section 2.5, Risk Management, and Software Configuration Control in the Software Configuration Management KA.)

A managed-change approach can also form the basis for evaluating success during closure of an incremental cycle or an entire project, based on changes that occurred along the way. (See Topic 5, Closure).

2. Software Project Planning

A key step in software project planning should be selecting an appropriate SDLC model and, perhaps, tailoring it based on project scope, software requirements and a risk assessment. The SWX [2] states that project life cycles occupy a continuum from predictive to adaptive. Factors that characterize the positions of software project life cycles within the continuum include (but are not limited to) the various ways requirements and plans are handled, how risk and cost are managed, and key stakeholder involvement. Highly predictive software project life cycles emphasize requirements specification and detailed planning during the project's initiation and planning phases. Detailed plans based on a known architecture, requirements and constraints are used to reduce risk and cost. Milestones are planned, versus continuous

key stakeholder involvement. Highly adaptive software project life cycles, on the other hand, are characterized by progressive requirements specification based on short iterative development cycles. Risk and cost are reduced by progressive evolution of initial plans, and key stakeholders are continuously involved [2].

Other factors to consider include the nature of the application domain, functional and technical complexity, and software quality requirements. (See Software Quality Requirements in the Software Quality KA.)

In all SDLCs, risk assessment should be an element of initial project planning, and the “risk profile” of the project should be discussed and accepted by all relevant stakeholders. Software quality management processes (see Software Quality Management Processes in the Software Quality KA) should be planned along with project planning. This planning should establish procedures and responsibilities for software quality assurance (SQA), verification and validation, reviews, and audits. (See the Software Quality KA.) Processes and responsibilities for ongoing review and revision of the project plan and related plans should also be clearly stated and agreed upon.

2.1. Process Planning [3, c3, c4, c5], [5*, c1]*

SDLC models span a continuum from predictive to adaptive. (See Software Life Cycle Models in the Software Engineering Process KA.) Predictive SDLCs are characterized by the development of detailed software architecture and software requirements, detailed project planning, and minimal planning for iteration among development phases. Adaptive SDLCs are designed to accommodate emergent software requirements and iterative adjustment of plans. A highly predictive

SDLC executes the first five processes listed in Figure 1 in a linear sequence with revisions to earlier phases only as necessary. Adaptive SDLCs are characterized by iterative development cycles. SDLCs in the midrange of the SDLC continuum produce increments of functionality on either a preplanned schedule (on the predictive side of the continuum) or as the products of frequently updated development cycles (on the adaptive side of the continuum).

Well-known SDLCs include the waterfall, incremental and spiral models, plus various Agile software development approaches [2, 11] [3*, c2].

Relevant methods (see the Software Engineering Models and Methods KA) and tools should be selected as part of planning. Automated tools that will be used throughout the project should also be planned for and acquired. Tools might include those for project scheduling, software requirements, software design, software construction, software maintenance, software configuration management, software engineering process and software quality, among others. Many of these tools should be selected based primarily on the technical considerations discussed in other KAs, but some of those concerns are closely related to the management considerations discussed in this chapter.

2.2. Determine Deliverables [3, c4, c5, c6]*

Each project activity’s work product(s) (e.g., software architecture design documents, inspection reports, tested software) should be identified and characterized. Opportunities to reuse software components from previous projects or to use off-the-shelf software products should be evaluated. Software procurement and use of third parties to develop deliverables should be

planned and suppliers selected. (See Section 3.2, Software Acquisition and Supplier Contract Management.)

2.3. Effort, Schedule and Cost Estimation

The topic of estimation in general is addressed in the Software Engineering Economics KA. Questions like “What is estimation?” and “Why do we estimate?” are addressed there. This section addresses management-specific estimation topics.

Estimating costs for software projects is an error-prone process. The effort required for any given software project depends almost entirely on human elements: individuals’ experience and capabilities, team members’ interactions, and the culture of the software development environment. Dynamic environmental factors, such as rapid technology evolution, changing and emergent requirements, and the intangible nature of the product, also significantly affect cost management. Estimating costs when this much variability exists is difficult even when significant historical data exists. Software project managers should use multiple estimation approaches and then reconcile the differences among the estimates [3, 10, 11].

When data is available, the estimated range of effort required for a project, or parts of a project, can be determined using a calibrated estimation model based on historical size and effort data. It is best to also use bottom-up estimation techniques based on estimates from those who will accomplish the work and historical data based on similar projects [2]. Task dependencies can be established, and potential opportunities for completing tasks concurrently and sequentially can be identified and documented, using a Gantt chart, for example. In predictive SDLC

projects, the expected schedule of tasks, with projected start times, durations and end times, is typically produced during planning. In adaptive SDLC projects, an overall estimate of effort and schedule is typically developed from the initial understanding of the requirements, or, alternatively, constraints on overall effort and schedule may be specified and used to determine an initial estimate of the number of iterative cycles and estimates of effort and other resources allocated to each cycle.

Resource requirements (for example, people and tools needed) can usually be translated into cost estimates. The estimation of effort, schedule and cost is an iterative activity that should be negotiated and revised among affected stakeholders until consensus is reached on resources and time available for project completion. Program manager often uses a model that links four association role types: responsible, accountable, consulted, and informed (i.e., RACI) to facilitate this process. Responsible roles produce deliverables; accountable roles check the deliverables; consulted roles advise on tasks; and informed roles are kept informed throughout these processes. Project managers should constantly monitor stakeholder requirements and changes as they evolve to analyze their impact on the project cost and schedule. This is usually more important in Agile software development projects, where stakeholder requirements are dynamic because changes might occur rapidly as the project progresses.

2.4. Resource Allocation [3, c5, c10, c11]*

Equipment, facilities and people should be allocated to the identified tasks, including allocating responsibilities for completing various project elements and the overall project. A matrix that shows who is

responsible for, accountable for, consulted about and informed about each task can be used. Resource allocation is based on and constrained by the availability of resources and their optimal use, and by issues relating to personnel (e.g., productivity of individuals and teams, team dynamics, and team structures).

2.5. Risk Management [3, c9] [5*, c5]*

Risk and uncertainty are related but distinct concepts. Uncertainty results from a lack of information. Risk is effect of uncertainty on objectives that has negative (threats) or positive (opportunities) consequences on objectives. Uncertainty often creates risk and is characterized by the probability that an event having a positive outcome might occur.

Risk management entails identifying risk factors, analyzing probability and potential impact of each risk factor, prioritizing risk factors, and developing risk mitigation strategies to reduce the probability of a negative event and to minimize the negative impact if a risk factor becomes a problem. Risk management data can be used to represent the project risk profile; this data is often part of a risk register. A risk register is a document used as a risk management tool. It can be used to fulfill regulatory compliance, serving as a repository for all risks identified and for additional information about each risk [2]. Risk assessment methods (e.g., expert judgment, historical data, decision trees and process simulations) can sometimes be used to identify and evaluate risk factors.

Project abandonment conditions can also be determined with all relevant stakeholders. Software-unique aspects of risk, such as software engineers' tendency to add unneeded features or the risks related to software's intangible nature, can influence risk management for software projects.

Particular attention should be paid to managing risks related to software quality requirements such as safety or security [11]. (See the Software Quality KA.) Risk management should be done not only at the beginning of a project, but also at periodic intervals throughout the project life cycle.

2.6. Quality Management [3, c4] [4*, c24]*

According to the *PMBOK® Guide*, Project quality management includes the performing organization's processes and activities that determine quality policies, objectives and responsibilities so the project will satisfy the needs for which it was undertaken. This section discusses additional considerations for managing software project quality [1]. Software quality requirements for a software project and the associated work products should be identified, perhaps both quantitatively and qualitatively. Quality attributes of software include but are not limited to safety, security, reliability, availability, performance, ease of use and ease of modification. *SWX* Section 1.9 lists quality attributes that are important for software users (e.g., efficiency, safety, security, reliability, availability) and quality attributes that are important to software developers and maintainers (e.g., maintainability is important to those who provide sustainment services) [1]. ISO/IEC 25000 series of standards provides extensive lists of software quality attributes that align with different stakeholder needs [2]. This alignment is consistent with ISO/IEC/IEEE 15939 and Practical Software and Systems Measurement (PSM) [2, 9.11].

Large portions of system functionality are shifting from hardware to software to capitalize on the increased flexibility and speed of component delivery that software can provide. However, with these benefits come other challenges — for example, the

need for increased management of software quality requirements (e.g., cybersecurity) throughout the SDLC [11]. Thresholds for acceptable quality measurements should be set for each software quality requirement based on stakeholder needs and expectations. Procedures concerned with ongoing SQA and quality improvement throughout the development process and with verifying and validating the deliverable software product should also be specified during quality planning (e.g., technical reviews and inspections or demonstrations of completed functionality). (See the Software Quality KA.)

2.7. Plan Management [3, c4]*

Except for older predictive programs, documenting and managing formal plans are becoming less emphasized in managing most software projects. (e.g., documentation plans are rarely used, especially when MBSE is used for product data). The said, where they are used, plans should be developed and managed for software projects when change is expected. The magnitude of the planning effort and the plan's content should be determined partly by the risk of not developing the plan. The management of the project plan should itself be planned. Plans and processes selected for software development should be systematically monitored, reviewed, reported and, when appropriate, revised. Plans associated with supporting processes (e.g., documentation, software configuration management, and problem resolution) also should be managed. Reporting, monitoring and controlling a project should fit within the selected SDLC and the realities of the project. Plans should account for the various artifacts that will be used to manage the project.

Project managers of predictive life cycle software projects put substantial effort into

up-front development of the project plan and integration of subsidiary plans developed by support personnel from other organizational units (e.g., estimation specialists in the Project Management Office (PMO)).

In other types of programs (e.g., adaptive programs) where formal plans are not usually used, the emphasis should be on selecting and retaining project information useful in project control and future projects, and establishing strategy, policies, and procedures. For example, in adaptive programs, managers will usually spend less effort up front on developing detailed scope, cost and schedule plans. But significant effort is typically spent defining monitor and control processes, such as requirements traceability, to ensure coordination among the project members or teams as the emerging plans are implemented [2].

3. Software Project Enactment

During software project enactment (also known as *project execution*), plans are implemented, and the processes embodied in the plans are enacted. Throughout, there should be a focus on adherence to the selected SDLC processes, with an overriding expectation that adherence will satisfy stakeholder requirements and achieve the project's objectives. Fundamental to enactment are the ongoing management activities of monitoring, controlling and reporting.

3.1. Implementation of Plans [4, c2]*

Project activities should follow the project plan and supporting plans. Project activities use resources (personnel, technology and funding) and generate work products (software design, software code and software test cases).

3.2. Software Acquisition and Supplier Contract Management [3, c3, c4]*

Software acquisition and supplier contract management concern issues involved in contracting with customers of the software development organization who acquire the deliverable work products and with suppliers who supply products or services to the software engineering organization.

Software acquisition is common practice in software development projects, with integrated development environments (IDEs) and package libraries allowing software engineers to acquire third-party libraries with minimal steps, facilitating the assessment of risk, legality and suitability. However, software is no longer exclusively acquired as a shrink-wrapped product via a complex supply chain process and purchasing route. The ease of acquiring software has resulted in a common attack surface and led to security vulnerabilities. Organizations should consider introducing technical or procedural controls to minimize risk potentially exposed by unfiltered access to external library repositories.

The different software acquisition classes include commercial off-the-shelf (COTS) software — an existing product acquired “as is” from another software vendor, with applicable license terms; software developed exclusively for the organization by another party — typically contracted and sometimes a customization of COTS software; open source software — nominally free, although the organization may purchase enhanced support or maintenance and must review the license for restrictions on use; customer loaned software — typically to provide simulation or integration with another system element; software as a service (SaaS) — which might include software the organization rents to fulfill a particular need (for example, a cloud-based hosting, source control or development environment).

Software projects typically use different acquisition approaches to obtain the necessary software components. However, regardless of how the software components are obtained, the following activities should be performed: verifying that each component is complete, correct and consistent concerning the architectural design and software requirements for that component; integrating the components; verifying that the integrated components are correct, complete and consistent concerning the architectural design and the software requirements; and validating that the integrated components will satisfy their intended purpose when used in their intended operating environment.

Different acquisition approaches (for obtaining software components) require different approaches to managing the project. For example, custom development requires detailed planning for the numbers and skills of the software developers, organizing the development team(s), allocating requirements to the teams, specifying project metrics to be collected, monitoring progress, and applying corrective actions when actual progress does not agree with planned progress. Licensing components involves evaluating candidate components; selecting appropriate components; and negotiating terms, conditions, and delivery dates for the selected components.

This might involve selecting appropriate contracts, such as fixed price, time-and-materials, cost plus fixed fee, and cost-plus incentive fee. Agreements with customers and suppliers typically specify the scope of work and the deliverables. The agreements can also include special clauses, such as clauses establishing penalties for late delivery or no delivery, and intellectual property agreements that specify what the suppliers are providing and what the acquirer is paying for, plus what will be

delivered to and owned by the acquirer. For software developed by suppliers (both those internal to and those external to the software development organization), agreements commonly establish software quality requirements.

After the agreement has been put in place, executing the project in compliance with the terms of the agreement should be managed. (See Chapter 12, *Software Extension to the PMBOK® Guide (SWX)*, Software Procurement Management, for more information on this topic [2].)

3.3. Implementation of Measurement Process [3, c7]*

The measurement process should be enacted during the software project to ensure that relevant and useful data is collected. (See Sections 6.2, Plan the Measurement Process, and 6.3, Perform the Measurement Process.)

3.4. Monitor Process [3, c8]*

Adherence to the project plan and related plans should be assessed continually and at predetermined intervals. Outputs and completion criteria for each task should also be assessed. Deliverables should be evaluated for their required characteristics (for example, via inspections or by demonstrating working functionality). Effort expenditure, schedule adherence, costs to date, and resource use should be analyzed. The project risk profile (see Section 2.5, Risk Management) should be revisited, and adherence to software quality requirements should be evaluated (see Software Quality Requirements in the Software Quality KA).

Measurement data should be analyzed. (See Statistical Analysis in the Engineering Foundations KA.) Variance analysis should be conducted to determine deviation of actual from expected outcomes and values. This analysis might examine cost overruns,

schedule slippage or other measures. Outlier identification and analysis of quality and other measurement data should be performed (e.g., defect analysis). (See Software Quality Measurement in the Software Quality KA.) Risk exposures should be recalculated. (See Section 2.5, Risk Management.) These activities can enable problem detection and exception identification based on thresholds that have been exceeded. Outcomes should be reported as necessary or when thresholds have been exceeded. For example, the timely identification, mitigation, and resolution of software security vulnerabilities and weaknesses that exceed expectations can affect the system's security posture [11].

3.5. Control Process [3, c7, c8]*

Project monitoring activities provide the basis for making decisions. Where appropriate, and when the probability and impact of risk factors are understood, changes can be made to the project. This may take the form of corrective action (e.g., retesting certain software components). It might involve incorporating additional actions (e.g., deciding to use prototyping to assist in software requirements validation; see Prototyping in the Software Requirements KA). It might also entail revising the project plan and other project documents (e.g., the software requirements specification) to accommodate unanticipated events and their implications.

In some instances, the control process might lead to abandonment of the project. In all cases, the software development team should adhere to software configuration control and software configuration management procedures. (See the Software Configuration Management KA.) Decisions should be documented and communicated to all relevant parties, plans should be revisited

and revised when necessary, and relevant data should be recorded. (See Section 6.3, Perform the Measurement Process.)

3.6. Reporting [3, c11]*

Progress to date should be reported at specified and agreed-upon times both within the organization (e.g., to a project steering committee) and to external stakeholders (e.g., clients or users). Reports should focus on the information needs of the target audience as opposed to the detailed status reporting within the project team.

4. Review and Evaluation

At prespecified times and as needed, overall progress toward the stated objectives and satisfaction of stakeholder (user and customer) requirements should be evaluated. Similarly, assessments of the effectiveness of the software process, the personnel involved, and the tools and methods used should also be undertaken regularly and as circumstances demand.

4.1. Determining Satisfaction of Requirements [4, c8]*

Achieving stakeholder satisfaction is a principal goal of the software engineering manager. Progress toward this goal should be assessed periodically. Progress should be assessed upon achieving a major project milestone (e.g., completing software design architecture or completing a software technical review) or upon completion of an iterative development cycle that results in a product increment. Variances from software requirements should be identified, and appropriate actions should be taken.

As in the control process activity above (see Section 3.5, Control Process), software configuration control and software configuration management procedures should be followed (see the Software

Configuration Management KA). Decisions should be documented and communicated to all relevant parties; plans should be revisited and revised as necessary; and relevant data should be recorded (see Section 6.3, Perform the Measurement Process).

4.2. Reviewing and Evaluating Performance [3, c8, c10]*

Periodic performance reviews for project personnel can provide insight into the likelihood of adherence to plans and processes and possible areas of difficulty (e.g., team member conflicts). The various project methods, tools and techniques should be evaluated for effectiveness and appropriateness. The project's process should also be systematically and periodically assessed for relevance, utility and efficacy. Where appropriate, project changes should be made and managed.

5. Closure

An entire project, a major project phase or an iterative development cycle reaches closure when all the plans and processes have been enacted and completed. The criteria for project, phase or iteration success should then be evaluated. Once closure has been established, archival, retrospective and process improvement activities can be performed.

5.1. Determining Closure [1, s3.7, s4.6]

Closure occurs when the specified tasks for a project, a phase or an iteration have been completed and satisfactory achievement of the completion criteria has been confirmed. Software requirements can be confirmed as satisfied or not, and the degree of achieving the objectives can be determined. Closure processes should involve relevant stakeholders and document relevant

stakeholders' acceptance; any known problems should be documented.

5.2. Closure Activities [2, s3.7, s4.8]

After closure has been confirmed, project materials should be archived in accordance with stakeholder agreed-upon rules for archival methods, location and duration — possibly including destruction of sensitive information, software and the medium on which copies are resident. For example, these rules could require that during closure, all data is removed and destroyed from any devices that contain relevant information before physical disposal of the devices (e.g., the hard drives of personal computers, servers, mainframes, personal digital assistants (PDAs), routers, firewalls, switches, tapes, diskettes, CDs, DVDs, cell phones, printers, Universal Serial Bus (USB) data storage devices).

The organization's measurement database should be updated with relevant project data. A project, phase or iteration retrospective analysis should be undertaken so that issues, problems, risks and opportunities encountered can be analyzed. (See Topic 4, Review and Evaluation.) Lessons learned should be drawn from the project and fed into organizational learning and improvement endeavors.

6. Software Engineering Measurement

The importance of software engineering measurement for good management and engineering practices is widely acknowledged. (See Measurement in the Engineering Foundations KA.) Effective software engineering measurement has become one of the cornerstones of organizational maturity. Measurement can be applied to organizations, projects, processes and work products. This section focuses on applying measurement at the

levels of projects, processes and work products.

This section follows ISO/IEC/IEEE 15939 standard [6], which describes a process to define the activities and tasks necessary to implement a software measurement process. The standard also includes a measurement information model. This model in the PSM Continuous Iterative Development Measurement Framework report is also elaborated for SDLC approaches [9].

6.1. Establish and Sustain Measurement Commitment [7*, c1, c2]¹

- Establish measurement requirements. Each measurement endeavor should be guided by organizational objectives and driven by a set of measurement requirements established by the organization and the project (e.g., an organizational objective might be first to market).
- Establish the scope of measurement. The project team should establish the organizational unit to which each measurement requirement is to be applied. This might be a functional area, a single project, a single site or an entire enterprise. The temporal scope of the measurement effort should also be considered because the time series of some measurements might be required (e.g., to calibrate estimation models). (See Section 2.3, Effort, Schedule and Cost Estimation.)
- Establish the team's commitment to measurement. The commitment should be formally established, communicated and supported by resources.

¹ These two chapters can be downloaded free of charge from <http://www.psmsc.com/PSMBook.asp>.

- Commit measurement resources. An organization's commitment to measurement is an essential factor for success, as evidenced by the assignment of resources for implementing the measurement process. Assigning resources includes allocation of responsibility for the various tasks of the measurement process (such as analyst and librarian). Adequate funding, training, tools and support to conduct the process should also be allocated.

6.2. Plan the Measurement Process [7, c1, c2]¹*

- Characterize the organizational unit. The organizational unit provides the context for measurement, so the organizational context should be explicit, including the organization's constraints on the measurement process. The characterization can be stated in terms of organizational processes, application domains, technology, organizational interfaces and organizational structure.
- Identify information needs. Information needs are based on the organizational unit's goals, constraints, risks, and problems and may be derived from business, organizational, regulatory and/or product objectives. Stakeholders should identify, prioritize, document, communicate and review these needs.
- Select measures. Select candidate measures, with clear links to the information needs. Select measures based on the priorities of the information needs and other criteria such as cost of collection; degree of

process disruption during collection; ease of obtaining accurate, consistent data; and ease of analysis and reporting. Internal quality characteristics (see Models and Quality Characteristics in the Software Quality KA) are often not contained in the contractually binding software requirements. Therefore, consider measuring the software's internal quality to provide an early indicator of potential issues that might affect external stakeholders.

- Define data collection, analysis and reporting procedures. This encompasses collection procedures and schedules, storage, verification, analysis, reporting and data configuration management.
- Select criteria for evaluating the information products. The organizational unit's technical and business objectives influence evaluation criteria. Information products include those associated with the product produced and those associated with the processes used to manage and measure the project.
- Provide resources for measurement tasks. The appropriate stakeholders should review and approve the measurement plan to include all data collection procedures; storage, analysis, and reporting procedures; evaluation criteria; schedules; and responsibilities. Criteria for reviewing these artifacts should be established at the organizational unit level or higher and should be used as the basis for these reviews. Such criteria should consider experience, resource availability and potential

disruptions to projects when changes from current practices are proposed. Approval demonstrates commitment to the measurement process.

- Identify resources to be made available for implementing the planned and approved measurement tasks. Resource availability may be staged in cases where changes are piloted before widespread deployment. Consider the resources necessary for successful deployment of new procedures or measures.
- Acquire and deploy supporting technologies. This includes evaluating available supporting technologies, selecting the most appropriate technologies, acquiring those technologies and deploying those technologies.

6.3. Perform the Measurement Process [7, c1, c2]*

Integrate measurement procedures with relevant software processes. The measurement procedures, such as data collection, should be integrated into the software processes they measure. This might involve changing current software processes to accommodate data collection or generation activities. It might also involve analyzing current software processes to minimize additional effort and evaluating the effect on employees to ensure acceptance of the measurement procedures. Consider morale issues and other human factors. In addition, communicate the measurement procedures to those providing the data. Training and support might also be needed. Data analysis and reporting

procedures are typically integrated similarly into organizational and project processes.

Collect data. Data should be collected, verified and stored. Collection can sometimes be automated by using SEM tools (see Topic 7, Software Engineering Management Tools) to analyze data and develop reports. Data may be aggregated, transformed or recorded as part of the analysis, using a degree of rigor appropriate to the nature of the data and the information needs. This analysis typically produces graphs, numbers or other indicators that inform conclusions and recommendations to present to stakeholders. (See Statistical Analysis in the Engineering Foundations KA.) The results and conclusions are reviewed using the organization's formal or informal process. Data providers and measurement users should participate in reviewing the data to ensure it is meaningful and accurate and can result in reasonable actions.

Communicate results. Document and communicate information products to users and stakeholders.

6.4. Evaluate Measurement [7, c1, c2]*

Evaluate information products and the measurement process against specified evaluation criteria, and determine the strengths and weaknesses of the information products or process. An internal process or an external audit can be used to perform the evaluation, including feedback from measurement users. Record lessons learned in an appropriate database.

Identify potential improvements. Such improvements might be changes in the format of indicators, changes in units measured or reclassification of measurement categories. Determine potential

improvements' costs and benefits, and report appropriate improvement actions.

Communicate proposed improvements to the measurement process owner and stakeholders for review and approval. Also, communicate the lack of potential improvements if the analysis fails to identify any.

7. Software Engineering Management Tools [3*, c5, c6, c7]

SEM tools are often used to provide visibility and control of SEM processes. Some tools are automated, whereas others are manually implemented. In addition, there has been a recent trend toward using integrated suites of software engineering tools throughout a project to plan, collect and record, monitor and control, and report project and product information. Tools can be divided into the following categories:

Project planning and tracking tools. Project planning and tracking tools can be used to estimate project effort and cost and to prepare project schedules. For example, some projects use automated estimation tools that use a software product's estimated size and other characteristics as input and then estimate the required total effort, schedule and cost. Planning tools also include automated scheduling tools that analyze the WBS tasks, their estimated durations, their precedence relationships and the resources assigned to each task to produce a Gantt chart.

Tracking tools can be used to track project milestones, regularly scheduled project status meetings, scheduled iteration cycles, product demonstrations and action items.

Risk management tools. Risk management tools (see Section 2.5, Risk Management) can be used to track risk identification, analysis and monitoring. These tools include simulation or decision trees to analyze the effect of costs versus payoffs and subjective estimates of the probabilities of risk events. For example, Monte Carlo simulation tools can be used to produce probability distributions of effort, schedule and risk by algorithmically combining multiple input probability distributions.

Communication tools. Communication tools can help provide timely and consistent information to relevant stakeholders involved in a project. Examples of such tools are email notifications and broadcasts to team members and stakeholders; regular communications of meeting minutes; and charts showing progress, backlogs, and maintenance request resolutions.

Measurement tools. Measurement tools support activities related to the software measurement program. (See Topic 6, Software Engineering Measurement.) There are few completely automated tools in this category. Measurement tools to gather, analyze and report project measurement data may be based on spreadsheets developed by project team members or organizational employees.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	Fairley 2009 [3*]	Sommerville 2011 [4*]	Boehm and Turner 2003 [5*]	McGarry et al. 2001 [7*]
1. Initiation and Scope Definition				
1.1. Determination and Negotiation of Requirements	c3			
1.2. Feasibility Analysis		c4		
1.3. Process for the Review and Revision of Requirements	c3			
2. Software Project Planning				
2.1. Process Planning	c2, c3, c4, c5		c1	
2.2. Determine Deliverables	c4, c5, c6			
2.3. Effort, Schedule and Cost Estimation	c6			
2.4. Resource Allocation	c5, c10, c11			
2.5. Risk Management	c9		c5	
2.6. Quality Management	c4	c24		
2.7. Plan Management	c4			
3. Software Project Enactment				
3.1. Implementation of Plans		c2		
3.2. Software Acquisition and Supplier Contract Management	c3, c4			
3.3. Implementation of Measurement Process	c7			
3.4. Monitor Process	c8			
3.5. Control Process	c7, c8			
3.6. Reporting	c11			
4. Review and Evaluation				
4.1. Determining Satisfaction of Requirements				
4.2. Reviewing and Evaluating Performance	c8, c10			
5. Closure				
5.1. Determining Closure				
5.2. Closure Activities				
6. Software Engineering Measurement				
6.1. Establish and Sustain Measurement Commitment				c1, c2
6.2. Plan the Measurement Process				c1, c2
6.3. Perform the Measurement Process				c1, c2
6.4. Evaluate Measurement				c1, c2
7. Software Engineering Management Tools	c5, c6, c7			

FURTHER READING

A Guide to the Project Management Body of Knowledge (PMBOK® Guide) [1].

The *PMBOK® Guide* provides guidelines for managing individual projects and defines project management-related concepts. It also describes the project management life cycle and its related processes, and the project life cycle. It is a globally recognized guide for the project management profession.

Software Extension to the Project Management Body of Knowledge (PMBOK® Guide) [2].

SWX provides adaptations of and extensions to the generic practices of project management documented in the *PMBOK® Guide* for managing software projects. The primary contribution of this extension to the *PMBOK® Guide* is a description of processes for managing adaptive life cycle software projects.

IEEE Standard Adoption of ISO/IEC 15939 [6].

This international standard identifies a process that supports defining suitable measures to address specific information needs. It identifies the activities and tasks necessary to successfully identify, define, select, apply and improve measurement within an overall project or organizational measurement structure.

J. McDonald, *Managing the Development of Software Intensive Systems*, Wiley, 2010 [8].

This textbook introduces project management for beginning software and hardware developers, plus unique advanced material for experienced project managers. Case studies are included for planning and managing verification and validation for large software projects and complex software and hardware systems, as well as

inspection results and testing metrics to monitor project status.

REFERENCES

- [1] Project Management Institute, *A Guide to the Project Management Body of Knowledge (PMBOK® Guide)*, 6th ed., Newton Square, PA: Project Management Institute, 2017.
- [2] *Software Extension to the Project Management Body of Knowledge (PMBOK® Guide)*, Fifth Edition, Project Management Institute, 2013.
- [3*] R. E. Fairley, *Managing and Leading Software Projects*. Hoboken, NJ: Wiley IEEE Computer Society Press, 2009.
- [4*] I. Sommerville, *Software Engineering*, 10th ed., New York: Addison-Wesley, 2015.
- [5*] B. Boehm and R. Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston: Addison-Wesley, 2003.
- [6] IEEE, *IEEE Standard Adoption of ISO/IEC 15939: 2007 Systems and Software Engineering Measurement Process*, ed: IEEE, 2017.
- [7*] J. McGarry, et al., *Practical Software Measurement: Objective Information for Decision Makers*, Addison-Wesley Professional, 2001.
- [8] J. McDonald, *Managing the Development of Software-Intensive Systems*. Hoboken, NJ: John Wiley and Sons, Inc., 2010.
- [9] *Practical Software and Systems Measurement Continuous Iterative Development Measurement Framework Parts 1-3: Concepts, Definitions, Principles, and Measures*, Version 2.1, April 15, 2021.

- [10] Dr. Sarah Sheard, Mickael Bouyaud, Macaulay Osaisai, Jeannine Siviy, and Dr. Ken Nidiffer. “*Book Club*”
- [11] Dr. Kenneth E. Nidiffer, Dr. Carol Woody, Timothy A. Chick, *Program Manager’s Guidebook for Software Assurance, Special Report*, CMU/SEI-2018-SR-025, Software Solutions and CERT Divisions, Software Engineering Institute/Carnegie Mellon University, August 2018.
- [12] Dr. R. E. Fairley, *Systems Engineering of Software-Enabled Systems*, ISBN 978-1-119-53501-0, 2019.
- [13] Defense Innovation Board, *Software Is Never Done: Refactoring the Acquisition Code for Competitive Advantage Defense*, v3.3, March 12, 2019.
- [14] “DevOps: Building Reliable and Secure Systems Including Application Build, Package, and Deployment,” *IEEE Standard*, 2675-2021, 2021.
- [15] Murali Chemuturi and Thomas Cagley, *Mastering Software Project Management: Best Practices, Tools and Techniques*, J. Ross Publishing, July 2010.

CHAPTER 10

SOFTWARE ENGINEERING PROCESS

ABBREVIATIONS

BPMN	Business Process Modeling Notation
CASE	Computer-aided software engineering
CMM	Capability Maturity Model
CMMI	Capability Maturity Model Integration
GQM	Goal-question-metric
IDEF0	Integration definition
KA	Knowledge Area
PDCA	Plan-Do-Check-Act
SLCM	Software life cycle model

SLCP	Software life cycle process
UML	Unified Modeling Language

INTRODUCTION

This chapter considers the software engineering process from several perspectives: concepts, life cycles, and software engineering process assessment. The software engineering community has been very active concerning the standardization of many of the aspects of the software engineering process.

BREAKDOWN OF TOPICS FOR THE SOFTWARE ENGINEERING PROCESS KA

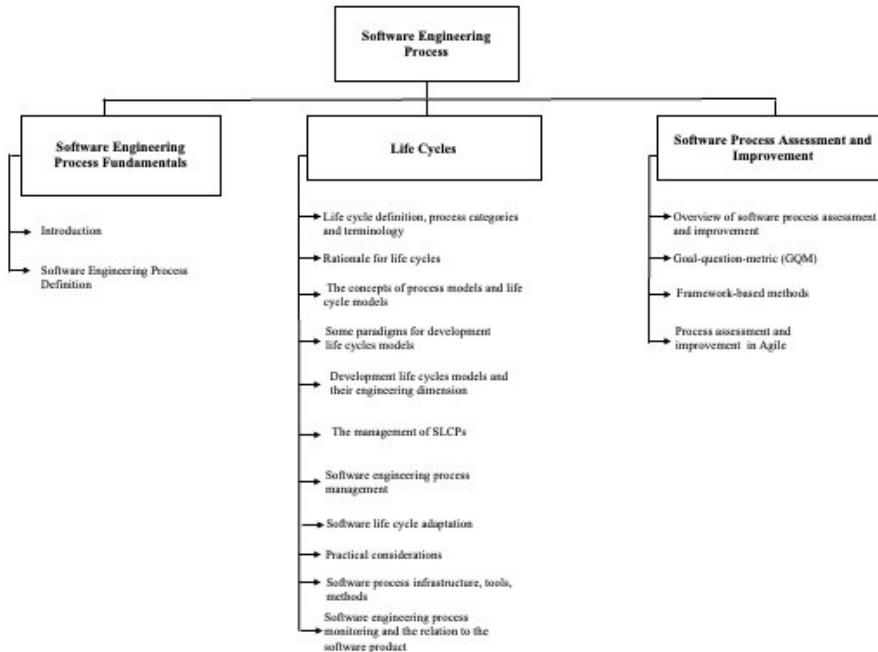


Figure 1. Breakdown of Topics for the Software Engineering Process KA

1. Software Engineering Process Fundamentals

1.1 Introduction

Software engineering processes involve work activities software engineers conduct to build and operate software. When the discipline of software engineering emerged, scientists, engineers and technicians had to look at existing disciplines to understand the scope of the software engineering process. An engineering process consists of a set of interrelated activities that transform one or more inputs into outputs while consuming resources to accomplish the transformation. As part of engineering, software engineering uses processes similar to those of other types of engineering. As engineers create devices or other products, they progress through various steps, expending significant design effort, relying on a vast trove of knowledge as they do so, at the time that they gain knowledge, i.e. learn, about the process they are performing and the product they are creating.

Beginning in the 1960s and continuing in the 1970s, engineering design and manufacturing provided a baseline — a foundation — for what would later become a new discipline. In those years, it was agreed that the process of building software would be decomposed into processes that could include design and manufacturing, and later, operations. Some of the processes needed to construct software systems fit into the design class, and others fit into the manufacturing class. Today, the software engineering community is still learning and, therefore, still improving the software engineering process. Currently, a wide consensus exists concerning that building software systems requires lots of design and learning effort focused on the product under construction, and on the process.

1.2 Software Engineering Process Definition

A process is a “set of interrelated or interacting activities that transforms inputs into outputs”,

The concept of a project emerges as an “endeavor with defined start and finish criteria undertaken to create a product or service in accordance with specified resources and requirements” [1] or a “temporary endeavor undertaken to create a unique product, service, or result” [13]. It is a concept of the management discipline linked to clear objectives and bound by a limited time frame, as discussed in Knowledge Area (KA) 9, Software Engineering Management. Software engineering processes are usually performed in the context of projects.

Many of the processes of the more conventional engineering disciplines (e.g., electrical or chemical) include design and manufacturing, where manufacturing produces multiple units of a system (e.g., a chemical reactor). This is not the case for software systems, though manufacturing is useful to describe the need to build the many software units that comprise a software system. In electrical or chemical engineering, the operation of the engineering systems transforms (raw) materials, energy and physical entities into other forms of material or energy. For the software engineering discipline, an analogy for this operation could be the execution of a software unit (the output from a software engineering set of processes) that transforms one kind of data into another.

In the rest of the section, the term *process* will denote work activities, not the execution of software.

The Software Engineering Process KA is closely related to this Software Engineering Process KA, and it includes Software Engineering Management, Software Engineering Models and Methods, Software Quality, Software Architecture, and Software Testing. The Measurement and Root Cause Analysis in the Engineering Foundations KA is also closely related.

where activity is a “set of cohesive tasks of a process,” and a task is a “required, recommended, or permissible action, intended to contribute to the achievement of

one or more outcomes of a process”[1]. According to [2], a process is a “predetermined course of events defined by its purpose or by its effect, achieved under given conditions.” A third definition, following [7], is a “system of activities, which use resources to transform inputs into outputs.” And a fourth one is a “set of interrelated or interacting activities which transforms inputs into outputs to deliver an intended result” [20]. That is, the description of a process includes required inputs, transforming activities, and the outputs

2. Life Cycles¹

2.1. Life cycle definition, process categories and terminology

A *life cycle*, according to [1], is the “evolution of a system, product, service, project or other human-made entity from conception through retirement.” In software engineering, life cycles help convey information about software systems, the “system[s] for which software is of primary importance to the stakeholders” [1]. The concept of life cycles was put in place because simply identifying and defining the processes required to produce software did not adequately describe all the complexity of software systems. It was also necessary to define life cycles, which include a number of processes and constraints [8].

In software engineering, *development* refers to a crucial stage of a system, product, service or project life cycle: that of building (or changing) a software system according to the stakeholders’ needs. From a production/industrial management perspective, software systems are referred to as products. In this context, the term *software product development lifecycle* makes sense.

Product life cycle can be defined as the “series of phases that represent the evolution of a product, from concept through delivery, growth, maturity, to retirement” [13]. This

generated. These definitions address any processes that are applied to the software part of software systems. Software systems also include hardware, and they also involve people and manual procedures. The output of one process can be an input to another process. Processes may include controls (e.g. directives and constraints) and enabling mechanisms (e.g. tools, technologies or resources such as workforce and infrastructure) associated with the processes [14].

definition is not specific to software systems but applies to all products more generally. Likewise, the life cycle concept, which is linked to the product concept, is not specific to software engineering.

Software systems contain software units that are an “atomic-level software component of the software architecture that can be subjected to stand-alone testing.” (See the Testing KA.) The life cycle of a software system (and keep in mind that software engineering uses an interdisciplinary approach) comprises all the processes, activities and tasks from the ideation of the software system to the retirement of the system, including production, operation and evolution, as well as acquisition, when needed, and supply. Likewise, we can look at the life cycle of an element of a software system (a software unit). A software system life cycle will consider both the business and the technical needs of the stakeholders and the system’s ability to produce, as the outcome of the different software life cycle processes (SLCPs) performed by a team, a product that meets the stakeholders’ needs, with the required quality level for its users and for all the different stakeholders.

The following paragraphs enumerate the process categories, as enumerated in [1]. These process categories reflect the multiple perspectives involved in producing a software

¹ Lifecycle, life-cycle and life cycle are different spellings. Merriam-Webster prefers the spelling “life cycle”.

system: (1) technical processes including engineering practices to build, make, evolve, operate and retire software products; (2) technical management processes that cover planning and control, as well as configuration management, risk management, information management and quality assurance; (3) organizational project-enabling processes that support life cycle model and infrastructure management, portfolio management, and human resources, knowledge and quality management; and finally (4) agreement processes, which are essential to support collective decision-making, as well as acquisition and supply processes.

A breakdown of these processes is as follows:

1) Technical processes

- a) Business or mission analysis process
 - b) Stakeholder needs and requirements definition process
 - c) System/software requirements definition process
 - d) Architecture definition process
 - e) Design definition process
 - f) System analysis process
 - g) Implementation process
 - h) Integration process
 - i) Verification process
 - j) Transition process
 - k) Validation process
 - l) Operation process
 - m) Maintenance process
 - n) Disposal process
- Technical management processes
- a) Project planning process

- b) Project assessment and control process
 - c) Decision management process
 - d) Risk management process
 - e) Configuration management process
 - f) Information management process
 - g) Measurement process
 - h) Quality assurance process
- 2) Organizational project-enabling processes
- a) Life cycle model management process
 - b) Infrastructure management process
 - c) Portfolio management process
 - d) Human resource management process
 - e) Quality management process
 - f) Knowledge management process
- 3) Agreement processes as well as acquisition and supply processes

2.2. Rationale for life cycles

Creating, operating and retiring software products require a number of processes, with their activities and tasks, and a number of constraints. As noted above, software systems involve people and manual procedures, as well as software and hardware. Defining software processes, following [12], requires specifying inputs and outputs. Inputs from processes are, very often, outputs from other processes. Therefore, life cycle processes are interrelated

processes; that is, each individual process (its inputs and outputs) may depend on other processes. The interrelated nature of the processes involved make the overall software engineering process highly complex.

The specification of life cycles is a powerful tool for implementing an engineering approach to the creation, operation and retirement of software systems. A life cycle should be defined following engineering principles that guide engineering as a discipline [8]. The specification of a life cycle includes the specification of every process and the associated constraints. The process specification should be useful to humans so that they can communicate with one another using this specification. The specification should be easy to understand and correct because life cycle specifications are the basis for technical and engineering management, including coordination and agreement, measurement, assessment and improvement, and quality management.

2.3. The concepts of process models and life cycle models

Section 2.1 provides a number of software life cycle definitions. In reference [2], a new definition introduces the concept of a *standard* as a commonly accepted guiding document, stating that a “project-specific sequence of activities ... is created by mapping the activities of a standard onto a selected software life cycle model (SLCM).” That is, a life cycle is created in conformance with the life cycle model.

Examples of well-known life cycle models for product development are, among others, the waterfall model, the V-model, the incremental model, the spiral model and the Agile model [2,3, 10].

2.4. Some paradigms for development life cycle models

Each software system has its own features reflecting the stakeholders’ needs, both business and technical. A suitable life cycle will

consider all these needs. As explained in Section 2.3, a software life cycle will be defined in conformance with (partially or fully conforming to) an SLCM. Some authors use the term “development” to refer to SLCM, e.g. “iterative development” instead of “iterative (software) life cycle model”. Types of life cycles are described below.

Predictive life cycles are “a form of project life cycle in which the project scope, time, and cost are determined in the early phase of the life cycle” [13]. Predictive life cycles assume that the set of requirements that will be implemented is a closed set that will not undergo substantive change unless a [force majeure](#) occurs.

An *iterative* life cycle is “a project life cycle where the project scope is generally determined early in the project life cycle, but time and cost estimates are routinely modified as the project team understanding of the product increases. Iterations develop the product through a series of repeated cycles, while increments successively add to the functionality of the product” [3, 8, 13]. The iterations duration is defined for each project. The method chosen (see KA 11) would prescribe the role and size of iterations.

In an *evolutionary* life cycle, a product or service changes over its lifetime. It may happen because requirements and customer needs change, but it also may happen because requirements are introduced into the product in successive steps and not as a complete and atomic set [3, 8]. “Successive steps” is a synonym for “iterations.”

An *incremental* life cycle is “an adaptive project life cycle in which the deliverable is produced through a series of iterations that successively add functionality within a predetermined time frame. The deliverable contains the necessary and sufficient capability to be considered complete only after the final iteration” [3, 8, 13]. Incremental life cycles are not always predictive, but they can be. *Incremental development* is a “software development technique in which requirements definition, design, implementation, and testing

occur in an overlapping, iterative (rather than sequential) manner, resulting in incremental completion of the overall software product” [2].

Continuous development refers to software engineering practices that allow for frequent releases of new systems (including software) to staging or various test environments through the use of automated tools [8, 9, 11].

A life cycle can enforce a rule that the requirements specifications cannot be changed once the requirements process has been finalized and the customer has agreed to the specifications. This happens, for example, in predictive life cycles. On the other hand, when the life cycle does not preclude changes in the requirements specifications, even after the customer has agreed to them and signed off on them, and in practice, allows them to change at any point [upon negotiation of interested parts], then the life cycle is said to be *open to change*. Being open to change is one of the claims of Agile development [9, 10].

2.5. Development life cycle models and their engineering dimension

Several life cycle models have become well known with the development of software engineering since its inception. One model, which became popular early in the history of the discipline, is the waterfall model [3], that falls into the category of *predictive*, described previously. The waterfall model approach for product development uses a number of phases, including requirements, preliminary design, detailed design, coding and testing. It implements a very strict process, in which one phase cannot be started until the previous one is finished. The waterfall model was useful because it introduced a systematization in the development of software systems and, therefore, what could be referred to as an engineering approach to software product development. Many variants or extensions, such as the V-model [3], with many different names and nuances, have been introduced in the history of software engineering. The waterfall model was an early attempt to

address the so-called software crisis [3]. The waterfall model is document-driven. Reference [2] defines the waterfall model as the “model of the software development process in which the constituent activities, typically a concept phase, requirements phase, design phase, implementation phase, test phase, and installation and checkout phase, are performed in that order, possibly with overlap but with little or no iteration.”

The waterfall model is clearly an example of a predictive life cycle. Some other paradigms, such as the incremental life cycle, also attempted to address the “software crisis.” In this model (see Section 2.4), different phases occur in an overlapping rather than sequential manner. An incremental life cycle can also be a predictive life cycle. This would mean that the requirements are defined and closed before any other development phase is started. The spiral model, introduced by Boehm, is evolutionary and risk-driven, as opposed to document- or code-driven [3]. Reference [2] defines the spiral model as a “model of the software development process in which the constituent activities, typically requirements analysis, preliminary and detailed design, coding, integration, and testing, are performed iteratively until the software is complete.” Another popular model is rapid prototyping, a “type of prototyping in which emphasis is placed on developing prototypes early in the development process to permit early feedback and analysis in support of the development process”[2].

The Agile Manifesto [16] effected a disruption in the software engineering community by creating an abrupt change of mindset. The difference was that Agile Manifesto signatories claimed that the process should be open to change — requirements could be modified at any stage of the development process if users’ needs changed. Communication and mutual trust between team/customer were essential. Signatories claimed that team communication, often face-to-face, and communication with the customer were key. Nevertheless, the Agile Manifesto does not say that documents (e.g. to define

requirements) are not needed, documents are needed [9, 10]. Signatories also advocated for small software incremental deliveries, as opposed to projects that applied the waterfall model with a single software delivery at the end of the project after months or years of working. Agile makes a clear distinction between, on the one side, values and principles (e.g., always delivering value to the customer or a commitment to technical excellence) and, on the other, practices (peer programming, sprint planning or retrospective). The Agile mindset [10] is different from the predictive mindset. The Agile mindset is based on a number of values and principles (e.g., the importance of communication, being open to change or commitment to technical excellence and always delivering value to the customer); this focus differentiates Agile from the predictive mindset, which is more focused on committing to the implementation of the requirements specifications. Agile helps address complexity [8, 10].

Several misconceptions arose around Agile, and some still remain. One is that Agile is a method in itself, which it is not. Another is that Agile is “faster” than waterfall because you need not produce any document. A third one is that Agile has a limited or unstructured set of methods/practices; a chart that enumerates several commonly used Agile methods and practices can be found, for example, in [18]. Several Agile methods became popular, like Extreme Programming for product development, Scrum for project management and others. Even considering the ever rising popularity of the Agile life cycle model to address complex projects, scaling up Agile for large projects and portfolios is still challenging. The perception today is that the Agile Manifesto meant a significant disruption; nevertheless, it is already 20 years old, and some authors think that some of its principles might need be updated, informed by the experience developers have obtained in the past 20 years [17].

The application of Agile practices has transcended the software engineering

process, and the terms *business agility* and *Agile organizations* are now very common [19]. From a software engineering point of view, Agile created an opportunity for the industry to achieve a reengineering and a better alignment of software engineering processes and business strategic processes in organizations. The use of an Agile approach by business processes is a common scenario; this is reflected in the principles of DevOps [11], for example, explained later in this section, and process assessment and improvement in Section 3.

The need to provide more frequent releases, the fact that users’ needs and technological life cycles change more frequently, together with the required alignment of the organizations’ strategic plans with the organizations’ IT operations, has led to the creation of DevOps, defined as a “set of principles and practices which enable better communication and collaboration between relevant stakeholders for the purpose of specifying, developing, and operating software and systems products and services, and continuous improvements in all aspects of the life cycle” [11]. The ability to provide more releases more frequently, once adequate process management has been defined, has become an advantage that makes companies more competitive.

In the history of software engineering, there has been a lot of controversy over software life cycle models — for example, as seen in debate over the merits of the waterfall model versus the Agile model of software development (See Section 2). This controversy should be understood from a historical perspective; new approaches have been disruptive or considered disruptive, and there has been a lack of empirical measures to support evidence-based discussions about software engineering. This situation has been changing slowly but steadily. Using empirical measures as the basis for making decisions is an essential element of software engineering [4, 8]. See also KA 9, Software Engineering Management, and KA 12, Software Quality.

2.6. The management of SLCPs

The life cycle for any software system contains a number of stages. According to [14], these stages are the following:

- 1) Concept: At this stage, stakeholders' needs will be identified, concepts will be explored, and solutions will be proposed.
- 2) Development: At this stage, requirements representing the users' needs will be refined, solutions will be created, systems built, and all undergo the needed verification and validation processes.
- 3) Production: This stage will have a different scope depending on the characteristics of the software system under focus. Generally speaking, it will include the production and testing of the system.
- 4) Utilization: At this stage, the system operates to satisfy users' needs.
- 5) Support: At this stage, developers provide the required actions to achieve a satisfactory operation.
- 6) Retirement: At this stage, the team follows established procedures to dispose of the system.

The stages are not supposed to be sequential, by any means. Actually, the specification of the life cycle for a software system will include the transitions between these stages. It should be clear that these stages have been identified for a general life cycle. Specific life cycles will have specific stages, meaningful to a particular project's stakeholders; these stages will fit into these general stages.

2.7. Software engineering process management

Process management is defined as "direction, control, and coordination of work performed to develop a product or perform a service" [2].

Several management levels govern the software engineering process, as explained in reference [1], see also KA 9, Software Engineering Management. The lowest level is the technical processes engineering management; the second is the technical management level, which is represented by the technical processes level and will include project management processes. The third level is the (executive) management level, focused on organizational enabling processes, such as knowledge management, life cycle model management, or portfolio management.

2.8. Software life cycle adaptation

Each software system has its differential characteristics. These differential characteristics, together with the stakeholders' needs, lead to specific life cycles. This adaptation will include identifying all the relevant characteristics, selecting the appropriate standards or documents internal to an organization, selecting a development strategy/life cycle model, stages, and processes, and documenting the decisions and rationale. The adaptation will not require keeping the names provided in Section 2, or including them all [5, 14, 23].

2.9. Practical considerations

Defining a life cycle process includes the specification of the four categories presented in Section 2. This means addressing technical processes (definition of the processes that will be required), organizational processes (this includes human resources, among other processes), technical management processes (how processes are related, how they are monitored and managed), and agreement processes.

The discipline of software engineering has been evolving since its conception for several reasons. The community has never stopped learning, while the complexity of the products has been ever-increasing. Defining a software life cycle for the development of a product requires considering the characteristics of the product (e.g., stakeholders' needs, product size or complexity) and others external to the

product, such as the stakeholders' characteristics. Something that the community has learned is that estimations and measurements are essential. Wrong or uncertain estimations in the context of a life cycle will lead to failure. Accurate estimations are not easy to produce.

A current trend in software engineering is a focus on continuous delivery, supported by realistic process and product estimations and measurements. A helpful lesson engineers have learned is that working with large processes without producing any deliverables along the way increases uncertainty. (See DevOps in Section 2.5.) The Agile mindset has contributed to this and has helped engineers recognize the importance of communication in the process.

When a project process is defined in conformity with a life cycle, it is important to make sure that it will be possible to have metrics/measure definitions that will result in realistic process (and product) estimations and measurements throughout project definition and execution, and to define the level of precision and uncertainty; project process (and product) measurements should always provide accurate information about what is happening (the status of the process and the product) while the life cycle process is executed. If we are uncertain about the accuracy of estimations and measurements, the project might not be successful. In this case, a reflection should take place on the overall approach. Historically, a lot of polemics have grown about the predictive life cycle versus the Agile life cycle. In software engineering, discussions should always be supported by realistic process and product estimations and measurements, which can accurately reduce the level of uncertainty.

2.10. Software process infrastructure, tools, methods

Several notations have been used for defining software processes, including natural language, specifying textual lists of constituent activities and tasks, data-flow diagrams, state charts, integration definition (IDEF0), Petri

nets, and Unified Modeling Language (UML) activity diagrams, and Business Process Modeling Notation (BPMN) [2, 3]. Software process infrastructure includes tools to support the definition of these processes (e.g., a BPMN toolkit) but mainly to support all specific processes (testing or configuration management). Process definitions will often include methods and formalism (e.g., Rational Unified Process or extreme programming) [3]. Tools will, ideally, have to support these methods and, as important, be integrated with them. Therefore, it is not enough that a tool supports testing. Once a code unit has been successfully tested, for example, this becomes useful information that should be recorded so that the rest of the team can be aware of this fact. This means that the configuration management tool and the testing tool will have to be integrated [3, 8]. The term *software engineering environment*, representing a set of integrated tools, is sometimes used. The term *CASE* (computer-aided software engineering) was popular in the 1980s and 1990s. Somehow, the power tools of the 1980s and 1990s were oversold as a cure for the software crisis. In any case, today, the automation of some processes (e.g., configuration management, or at least version control; testing; ticket management) is seen as essential for the implementation of successful life cycles. You can also read KA 11, Software Engineering Models and Methods.

2.11. Software engineering process monitoring and its relationship with the software product

Developers must monitor the software engineering process execution, assess whether the process objectives are met, and assess risks. This process monitoring is part of software engineering process assessment (see Section 3) and part of the Software Engineering Management KA [1, 3, 4, 8].

Empirical methods support process assessment and improvement as well as product assessment and improvement. The goal of process execution is to obtain products

that meet stakeholders' needs. While this area is focused on the software engineering process, process monitoring requires assessing both process and product, using a joint, more holistic approach.

3. Software Process Assessment and Improvement

3.1. Overview of software process assessment and improvement

The idea that any executed process can be improved was present in the classic Shewhart-Deming Plan-Do-Check-Act (PDCA) paradigm [15, 24], which was already being discussed and applied in the 1950s, and its foundations can be found centuries earlier. For the software engineering process, several approaches have been developed.

The PDCA paradigm is an opportunity to meet a widely recognized need — the need for empirical evidence to make decisions. Such decisions include choosing a life cycle, deciding how to assess a process or deciding how to improve a process, among others. Getting empirical evidence across the execution of a software engineering process is essential for the success of the process execution.

3.2. Goal-question-metric (GQM)

The GQM approach [21] is based on Basili's Quality Improvement Paradigm. Both are based on setting goals that can be measured, changing something, and then evaluating the effect of the change. When the evaluation is positive, an improvement has occurred.

3.3 Framework-based methods

Some assessment methods are based on frameworks that use a process reference model and an assessment reference model — for example, CMM (Capability Maturity Model), CMMI [4, 22], and the ISO/IEC 33000 [4, 6] series, also known as SPICE.

The ISO/IEC 33000 framework includes a process reference framework and a process assessment model. The ISO/IEC 33000 framework revises the ISO/IEC 15504 series of International Standards, providing a framework for the assessment of (1) the process quality characteristics, one of which is process capability, together with (2) organizational maturity. This framework covers processes for the development, maintenance and use of systems across the information technology domain, as well as processes for the design, transition, delivery and improvement of services. The concept of seeking continuous improvement underlies the assessment.

This series has developed several groups of standards addressing, as well as core elements, basic requirements for performing process assessments, process models and the process measurement framework; guidance on how to perform assessments; measurement frameworks for the assessment of process capability and organizational maturity; process reference models for special cases such as safety or high maturity; process assessment models for SLCPs, system life cycle process IT service management, safety and high maturity; and organizational maturity models.

The process reference model is defined as a "model comprising definitions of processes in a domain of application described in terms of process purpose and outcomes, together with an architecture describing the relationships between the processes." The process assessment model is defined as a "model suitable for the purpose of assessing a specified process quality characteristic, based on one or more process reference models."

3.4. Process assessment and improvement in Agile

Agile methods (e.g., the scrum project management method) introduce what they call *retrospectives* at the end of each iteration. The objective of the retrospective is to analyze

what went well and what did not go well, to understand why, and to set a number of actions for learning and improvement. In the

end, the team is in a continuous learning loop [9].

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	ISO/IEC/I EEE 12207 [1*]	Sommervi lle [3*]	Laport et al.[4*]	Farley [8*]	Shore et al [9*]	PMI [10*]	Others
Software Engineering Process Fundamentals	c 5	c2		c1			
Introduction							[13]
Software Engineering Process Definition	c5	c2					[2] [7][14] [20]
Life Cycles	c 5-6	c2-3		c1-3			
Life cycle definition, process categories and terminology	c5-6	c2		c1-3			[13]
Rationale for life cycles				c2-3			[12]
The concept of process models and life cycles models		c2				c2	[2]
])Some paradigms for development life cycle models		c2-3		c2-3	c1	c1	[2] [13]
Development life cycle models and their engineering dimension		c2		c2-3	c1	c1	[2] [11] [16] [17] [18] [19]
The management of SLCPs							[14]
Software engineering process management	c5						[2]
Software life cycle adaptation							[5] [14] [23]
Practical considerations		c2		c2-3			[2]
Software process infrastructure, tools, methods		c2		c2-3			[2]
Software engineering process monitoring	c5-6	c2	c4-10	c2-3			

and its relationship with the software product							
Software Process Assessment and Improvement			c4-10				
Overview of software process assessment and improvement			c4				[15] [24]
Goal-question metric (GQM)							[21]
Framework-based methods			c4-10				[6] [22]
Process Assessment and improvement in Agile					c11		

REFERENCES

- [1] ISO/IEC/IEEE 12207 *Systems and software engineering — Software life cycle processes.*
- [2] ISO/IEC/IEEE 24765 Systems and software engineering — Vocabulary.
- [3] Ian Sommerville, *Software Engineering*. 10th ed. 2016.
- [4] C. Y. Laporte, A. April, *Software Quality Assurance*, IEEE Computer Society Press, 1st ed., 2018.
- [5] Project Management Institute, *Software Extension to the PMBOK® Guide — Fifth Edition*, 2013.
- [6] ISO/IEC 33001:2015 Information technology — Process assessment — Concepts and terminology.
- [7] ISO/IEC 25000:2014 Systems and software engineering — Systems and software product quality requirements and evaluation (SQuaRE) — Guide to SQuaRE.
- [8] David Farley, *Modern Software Engineering: Doing What Works to Build Better Software Faster*. Addison-Wesley Professional, December 2021.
- [9] James Shore, Shane Warden, *The Art of Agile Development*, O'Reilly Media, 2nd ed. October 2021.
- [10] Project Management Institute, *Agile Practice Guide*. Project Management Institute and Agile Alliance. September 2017.
- [11] IEEE 2675-2021 IEEE standard for DevOps: Building reliable and secure systems, including application build, package, and deployment.
- [12] ISO/IEC/IEEE 24774:2021 Systems and software engineering — Life cycle management — Specification for process description.
- [13] Project management Institute, *A Guide to the Project Management Body of Knowledge (PMBOK® Guide) — Sixth Edition.*
- [14] ISO/IEC/IEEE 24748-1:2018(E) Systems and software engineering — Life cycle management — Part 1: Guidelines for life cycle management.
- [15] Shewhart, W. A., Deming, W. E. . *Statistical Method from the Viewpoint of Quality Control*. Dover, New York. (1986)
- [16] “The Agile Manifesto.” <https://agilemanifesto.org>. [Accessed March 5, 2022].
- [17] Steve McConnell. *More Effective Agile: A Roadmap for Software Leaders*. 2019.
- [18] “Subway Map to Agile Practices.” Agile Alliance. <https://www.agilealliance.org/agile101/subway-map-to-agile-practices/>. [Accessed March 5, 2022].
- [19] Jutta Eckstein, John Buck. Company-wide Agility with Beyond Budgeting, Open Space & Sociocracy: Survive & Thrive on Disruption. 2021.
- [20] ISO/IEC TR 29110-5-3:2018 Systems and software engineering — Lifecycle profiles for very small entities (VSEs) — Part 5-3: Service delivery guidelines.
- [21] Norman Fenton, James Bieman. *Software Metrics*, 3rd ed. CRC Press, 2014.
- [22] CMMI Institute — CMMI V2.0. <https://cmmiinstitute.com/cmmi>. [Accessed March 5, 2022].
- [23] ISO/IEC/IEEE 24748-3:2020. Part 3: Guidelines for the application of ISO/ IEC/IEEE 12207 (software life cycle processes)
- [24] D.R. Kiran. *Total Quality management*. Elsevier , 2017

CHAPTER 11

SOFTWARE ENGINEERING MODELS AND METHODS

ACRONYMS

3GL	3rd Generation Language
BNF	Backus-Naur Form
FDD	Feature-Driven Development
IDE	Integrated Development Environment
PBI	Product Backlog Item
RAD	Rapid Application Development
UML	Unified Modeling Language
XP	eXtreme Programming

INTRODUCTION

Software engineering models and methods impose structure on software engineering to make it systematic, repeatable and ultimately more success-oriented. Models provide an approach to problem-solving, a notation and procedures for model construction and analysis. Methods provide an approach to the systematic specification, design, construction, testing and verification of the end-item software and associated work products.

Software engineering models and methods vary widely in scope — from addressing a single software life cycle phase to covering the complete software life cycle. This knowledge area (KA) focuses on models and methods that encompass multiple

software life cycle phases, since other KAs cover methods specific to single life cycle phases .

BREAKDOWN OF TOPICS FOR SOFTWARE ENGINEERING MODELS AND METHODS

This chapter on software engineering models and methods is divided into four main topic areas:

- *Modeling* discusses the general practice of modeling and presents topics in modeling principles; properties and expression of models; modeling syntax, semantics and pragmatics; and preconditions, postconditions and invariants.
- *Types of Models* briefly discusses models and aggregation of submodels and provides general characteristics of model types commonly found in the software engineering practice.
- *Analysis of Models* presents common analysis techniques used in modeling to verify completeness, consistency, correctness, traceability and interaction.
- *Software Engineering Methods* presents a summary of commonly used software engineering methods. The discussion guides the reader through a summary of heuristic methods, formal methods, prototyping and A gile methods.

The breakdown of topics for the Software Engineering Models and Methods KA is shown in Figure 1.

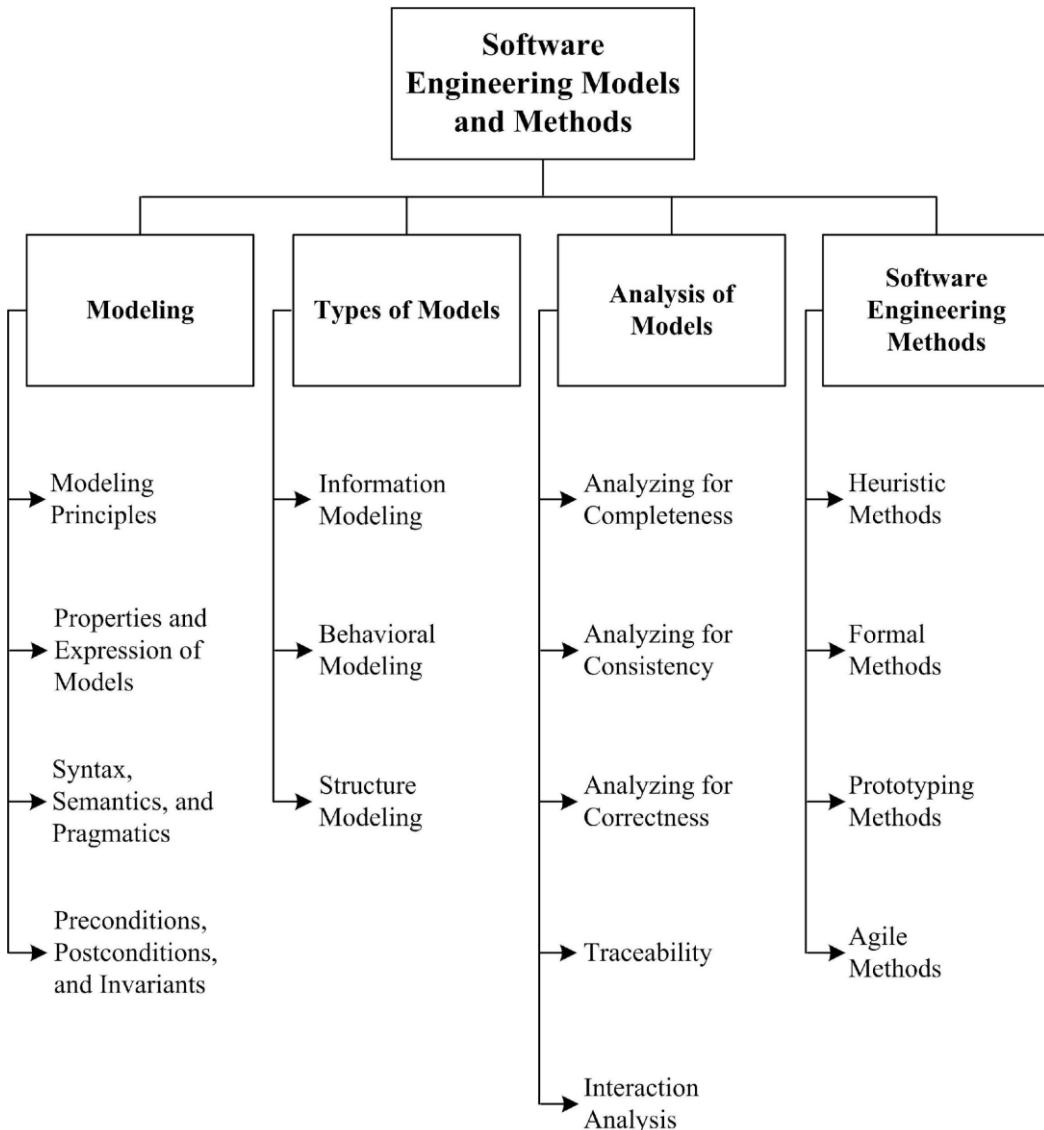


Figure 1. Breakdown of Topics for the Software Engineering Models and Methods KA

1. Modeling

Modeling of software is becoming a pervasive technique to help software engineers understand, engineer and communicate aspects of the software to appropriate stakeholders. *Stakeholders* are those people or parties with a stated or implied interest in the software (e.g., users, buyers, suppliers, architects, certifying authorities, evaluators, developers, software engineers).

Although there are many modeling languages, notations, techniques and tools in the literature and in practice, some general, unifying concepts apply to them all. The following sections provide background on these general concepts.

1.1. Modeling Principles [1*, c2s2, c5s1, c5s2, 2*, c2s2, 3*, c5s0]

Modeling provides the software engineer with an organized and systematic approach for representing significant aspects of the

software under study, facilitating decision-making about the software or elements, and communicating those significant decisions to others in the stakeholder communities. Three general principles guide such modeling activities:

- *Model the essentials*: Good models do not usually represent every aspect or feature of the software under every possible condition. Modeling typically involves only those aspects or features that pose specific questions, abstracting away any nonessential information. This approach keeps models manageable and useful.
- *Provide perspective*: Modeling provides views of the software under study using defined rules for expressing the model within each view. This perspective-driven approach provides dimensionality to the model (e.g., providing a structural view, a behavioral view, a temporal view, an organizational view and/or other views if relevant). Organizing information into views focuses the software modeling efforts on specific concerns relevant to that view using the appropriate notation, vocabulary, methods and tools.
- *Enable effective communications*: Modeling uses the application domain vocabulary of the software, a modeling language and semantic expression (in other words, meaning within context). When used rigorously and systematically, modeling results in a reporting approach that facilitates effective communication of software information to project stakeholders.

A model is an *abstraction* or simplification of a software component. A consequence of using abstraction is that, because no single abstraction completely describes a software component, the software model comprises an aggregation of abstractions,

which, when taken together, describe selected aspects, perspectives or views — only those that are needed to make informed decisions and respond to the reasons for creating the model in the first place. This simplification points to assumptions about the context within which the model is placed that should also be captured in the model. Then, when the model is reused, these assumptions can be validated first to establish the relevancy of the reused model within its new use and context.

1.2. Properties and Expression of Models [1*, c5s2, c5s3, 3*, c4s1.1p7, c4s6p3, c5s0p3]

Properties of models are those distinguishing features of a particular model that characterize its completeness, consistency and correctness within the chosen modeling notation and tooling. Properties of models include the following:

- *Completeness* — the degree to which all requirements have been implemented and verified within the model
- *Consistency* — the degree to which the model contains no conflicting requirements, assertions, constraints, functions or component descriptions
- *Correctness* — the degree to which the model satisfies its requirements and design specifications and is free of defects

Models are constructed to represent real-world objects and their behaviors to answer specific questions about how the software is expected to operate. Interrogating the models — through exploration, simulation or review — might expose areas of uncertainty within the model and the software to which the model refers. These uncertainties or unanswered questions regarding the requirements, design and/or implementation can then be handled appropriately.

The primary expression element of a model is an *entity*. An entity may represent concrete artifacts (e.g., processors, sensors or robots) or abstract artifacts (e.g., software modules or communication protocols). Model entities are connected to other entities using relations (lines or textual operators on target entities). Expression of model entities may be accomplished using textual or graphical modeling languages; both modeling language types connect model entities through specific language constructs. The meaning of an entity may be represented by its shape, its textual attributes or both. Generally, textual information adheres to language-specific syntactic structure. The precise meanings related to the modeling of context, structure or behavior using these entities and relations are dependent on the modeling language used, the design rigor applied to the modeling effort, the specific view being constructed and the entity to which the specific notation element may be attached. Multiple views of the model may be required to capture the needed semantics of the software.

When using automation-supported models, models may be checked for completeness and consistency. The usefulness of these checks depends greatly on the level of semantic and syntactic rigor applied to the modeling effort and on explicit tool support. Correctness is typically checked through simulation and/or review.

1.3. Syntax, Semantics and Pragmatics [2*, c2s2.2.2p6, 3*, c5s0]

Models can be surprisingly deceptive. The fact that a model is an abstraction with missing information can give people the illusion that they completely understand the software after studying a single model. A *complete model* (“complete” being relative to the modeling effort) may be a union of multiple submodels and any special function models. Examination of and decision-making

regarding a single model within this collection of submodels may be problematic.

Understanding the precise meanings of modeling constructs can also be difficult. Syntactic and semantic rules define modeling languages. For textual languages, *syntax* is defined using a notation grammar that defines valid language constructs (e.g., Backus-Naur form (BNF)). For graphical languages, syntax is defined using graphical models called *metamodels*. As with BNF, metamodels define a graphical modeling language’s valid syntactical constructs. In addition, the metamodel defines how these constructs can be composed to produce valid models.

Semantics for modeling languages specify the meaning attached to the entities and relations captured within the model. For example, a simple diagram of two boxes connected by a line is open to various interpretations. Knowing that the diagram on which the boxes are placed and connected is an object diagram or an activity diagram can assist in interpreting this model.

As a practical matter, the semantics of a specific software model are usually fairly clear due to the model’s use of a modeling language with consistent rules, the way that modeling language expresses entities and relations within that model, the experience of the modeler(s), and the context within which the modeling has been undertaken and represented. Meaning is communicated through the model even in the presence of incomplete information through abstraction. *Pragmatics* explains how meaning is embodied in the model and its context and how it is communicated effectively to other software engineers.

However, there are still instances where caution is needed regarding modeling and semantics. For example, any model parts imported from another model or library must be examined for semantic assumptions that

conflict with the new modeling environment; these conflicts might not be obvious. The model should be checked for documented assumptions. Although the employed modeling syntax might be the same, it might mean something quite different in the new environment, which is a different context. Also, consider that as software matures and changes are made, semantic discord can be introduced, leading to errors. With many software engineers working on part of a model over time, and with tool updates and perhaps new requirements, there are opportunities for portions of the model to represent something different from the original author's intent and initial model context.

1.4. Preconditions, Postconditions and Invariants [2*, c4s4, 4*, c10s4p2, c10s5p2p4]

When modeling functions or methods, the software engineer typically starts with assumptions about the software's state before, during and after the function or method executes. These assumptions are essential to the correct operation of the function or method and are grouped, for discussion, as a set of preconditions, postconditions and invariants.

- *Preconditions* are conditions that must be satisfied before execution of the function or method. If these preconditions do not hold before execution of the function or method, the function or method might produce erroneous results.
- *Postconditions* are conditions guaranteed to be true after the function or method has executed successfully. Typically, the postconditions represent how the software's state has changed, how parameters passed to the function or method have changed, how data values

have changed, or how the return value has been affected.

- *Invariants* are conditions within the operational environment that persist (in other words, do not change) before and after execution of the function or method. These invariants are relevant and necessary to the software and to the correct operation of the function or method.

2. Types of Models

A typical model consists of an aggregation of submodels. Each submodel is a partial description and is created for a specific purpose . A submodel may comprise one or more diagrams. The collection of submodels may use multiple modeling languages or a single modeling language. The Unified Modeling Language (UML) recognizes a rich collection of modeling diagrams. These diagrams, along with the modeling language constructs, are used in three common model types : information models, behavioral models and structure models. (See Section 1.1.)

2.1. Information Modeling [1*, c7s2.2, 3*, c8s1]

Information models focus on data and other information. An *information model* is an abstract representation that identifies and defines a set of concepts, properties, relations and constraints on data entities. The semantic or conceptual information model is often used to provide some formalism and context to the software as viewed from the problem perspective, without concern for how this model is mapped to the implementation of the software. The semantic or conceptual information model is an abstraction and, as such, includes only the concepts, properties, relations and constraints needed to conceptualize a real-world view of the information. Subsequent transformations of the semantic or conceptual

information model become logical and then physical data models as implemented in the software.

2.2. Behavioral Modeling [1*, c7s2.1, c7s2.3, c7s2.4, 2*, c9s2, 3*, c5s4]

Behavioral models identify and define software functions . Behavioral models generally take three basic forms: state machines, control-flow models and data-flow models. State machines provide a model that represents the software as a collection of defined states, events and transitions. The software transitions from one state to the next through a guarded or unguarded triggering event that occurs in the modeled environment. Control-flow models depict how a sequence of events causes processes to be activated or deactivated. Data-flow behavior is typified as a sequence of steps where data moves through processes toward data stores or data sinks.

2.3. Structure Modeling [1*, c7s2.5, c7s3.1, c7s3.2, 3*, c5s3, 4*, c4]

Structure models illustrate software's physical or logical composition from its various parts. Structure modeling establishes the defined boundary between the software being implemented or modeled and the environment in which it is to operate. Some common structural constructs used in structure modeling are composition, decomposition, generalization and specialization of entities; identification of relevant relations and cardinality between entities; and the definition of process or functional interfaces. Structure diagrams provided by the UML for structure modeling include class, component, object, deployment and packaging diagrams.

3. Analysis of Models

The development of models allows the software engineer to study, reason about and understand software structure,

function, operational use and assembly considerations . Analysis of constructed models is needed to ensure that the models are complete, consistent and correct enough to serve their intended purpose for the stakeholders.

The following sections briefly describe the analysis techniques generally used to ensure that the software engineer and other relevant stakeholders gain appropriate value from the development and use of models.

3.1. Analyzing for Completeness [3*, c4s1.1p7, c4s6, 5*, pp8-11]

To ensure software fully meets the needs of the stakeholders, testing for completeness — from the requirements elicitation process to code implementation — is critical. *Completeness* is the degree to which all specified requirements have been implemented and verified. Engineers can check models for completeness with a modeling tool that uses structural analysis and state-space reachability analysis (which ensure some set of correct inputs reach all paths in the state models). Models may also be checked manually for completeness by using inspections or other review techniques. (See the Software Quality KA.) Errors and warnings generated by these analysis tools and found by inspection or review indicate what corrective actions are probably needed to ensure model completeness .

3.2. Analyzing for Consistency [3*, c4s1.1p7, c4s6, 5*, pp8-11]

Consistency is the degree to which models contain no conflicting requirements, assertions, constraints, functions or component descriptions. Typically, consistency checking is accomplished with the modeling tool using an automated analysis function . Models may also be checked manually for consistency using

inspections or other review techniques. (See the Software Quality KA.) As with completeness, errors and warnings generated by these analysis tools and found by inspection or review indicate the need for corrective action.

3.3. Analyzing for Correctness [5, pp8-11]*

Correctness is the degree to which a model satisfies its software requirements and software design specifications, is free of defects, and ultimately meets the stakeholders' needs. Analyzing for correctness includes verifying the model's syntactic correctness (that is, correct use of the modeling language grammar and constructs) and semantic correctness (that is, use of the modeling language constructs to correctly represent the meaning of that which is being modeled). To analyze a model for syntactic and semantic correctness, one analyzes it — either automatically (e.g., using the modeling tool to check for model syntactic correctness) or manually (using inspections or other review techniques) — searching for possible defects and then removing or repairing the confirmed defects before the software is released for use.

3.4. Traceability [3, c4s7.1, c4s7.2]*

Developing software typically involves using , creating and modifying many work products such as planning documents, process specifications, software requirements, diagrams, designs and pseudo-code, handwritten and tool-generated code, manual and automated test cases and reports, and files and data. These work products may share various dependency relationships (e.g., uses, implements and tests). As software is developed, managed, maintained or extended, these traceability relationships must be mapped and controlled to demonstrate the software requirements' consistency with the software model (see

Requirements Tracing in the Software Requirements KA) and the many work products. Use of traceability typically improves the management of software work products and software process quality and assures stakeholders that all requirements are satisfied. Traceability enables change analysis once the software is developed and released because relationships to software work products can easily be traversed to assess change impact. Modeling tools typically help automatically or manually specify and manage traceability links among requirements, design, code and/or test entities that might be represented in the models and other software work products. (For more information on traceability, see the Software Configuration Management KA.)

3.5. Interaction Analysis [2, c10, c11, 3*, c29s1.1, c29s5, 4*, c5]*

Interaction analysis focuses on the communications or control-flow relations between entities used to accomplish a specific task or function within the software model. This analysis examines the dynamic behavior of the interactions among the software model's different portions , including other software layers (such as the operating system, middleware and applications).

Examining interactions between the computer software application and the user interface software might also be important for some software applications. Some software modeling environments provide simulation facilities to study aspects of the dynamic behavior of modeled software. Stepping through the simulation allows the software engineer to review the interaction design and verify that the software's different parts work together to provide the intended functions.

4. Software Engineering Methods

Software engineering methods provide an organized and systematic approach to developing software for a target computer. There are numerous methods from which to choose, and the software engineer needs to choose an appropriate method or methods for the software development task at hand. This choice can dramatically affect the success of the project. When software engineers, working with people who have the right skill sets and the right tools, use these software engineering methods, they can visualize the software's details and ultimately transform the representation into a working set of code and data.

Selected software engineering methods are discussed below. The topic areas are organized into discussions of Heuristic Methods, Formal Methods, Prototyping Methods and Agile Methods.

4.1. Heuristic Methods [1, c13, c15, c16, 3*, c2s2.2, c7s1, c5s4.1]*

Heuristic methods are experience-based software engineering methods that are fairly widely practiced in the software industry. This topic area contains three broad discussion categories: structured analysis and design methods, data modeling methods, and object-oriented analysis and design methods.

- *Structured analysis and design methods:* These methods develop the software model primarily from a functional or behavioral viewpoint. It starts from a high-level view of the software (including data and control elements). It then progressively decomposes or refines the model components through increasingly detailed designs. The detailed designs eventually converge to become a set of specific software details or specifications that must be coded (by hand, automatically generated or both), built, tested and verified.

- *Data modeling methods:* The data model is constructed from the viewpoint of the data or information used. Data tables and relationships define the data models. This data modeling method is used primarily to define and analyze data requirements supporting database designs or data repositories typically found in business software, where data is actively managed as a business systems resource or asset.
- *Object-oriented analysis and design methods:* The object-oriented model is represented as a collection of objects that encapsulate data and relationships and interact with other objects through methods. Objects may be real-world items or virtual items. The software model is constructed using diagrams to constitute selected views of the software. Progressive refinement of the models leads to a detailed design. The detailed design is then either evolved through successive iterations or transformed (using some mechanism) into the implementation view of the model, where the code and packaging for eventual software product release and deployment are expressed. (See Software Design KA, Model-B based Requirements and Software Requirements KA.)

4.2. Formal Methods [1, c18, 3*, c27, 5*, pp8-24]*

Formal methods are software engineering methods that apply rigorous, mathematically based notation and language to specify, develop and verify the software. Through use of a specification language, the software model can be systematically checked for consistency (or lack of ambiguity), completeness, and correctness, either automatically or semi automatically. This topic is related to the Formal

Analysis section in the Software Requirements KA.

This section addresses specification languages, program refinement and derivation, formal verification, and logical inference.

- *Specification languages*: Specification languages provide the mathematical basis for a formal method. Specification languages are formal, higher-level computer languages (not a classic 3rd-generation 1 language (3GL) programming language) used during the software specification, requirements analysis and/or design stages to describe specific input/output behavior. Specification languages are not directly executable languages. Instead, they typically comprise a notation and syntax, semantics for the use of the notation, and a set of allowed relations for objects.
- *Program refinement and derivation*: Program refinement creates a lower-level (or more detailed) specification using a series of transformations. Through successive transformations, the software engineer derives an executable representation of a program. Specifications may be refined, adding details until the model can be formulated in a 3GL programming language or in an executable portion of the chosen specification language. This specification refinement is made possible by defining specifications with precise semantic properties. For example, the specifications must set out not only the relationships between entities but also the exact runtime meanings of those relationships and operations.
- *Formal verification*: Model checking is a formal verification method. It typically involves performing a state-space exploration or reachability analysis to demonstrate that the represented software design has or preserves certain model properties of interest. An example of model checking is an analysis that verifies correct program behavior under all possible interleaving of event or message arrivals. Formal verification requires a rigorously specified model of the software and its operational environment. This model often takes the form of a finite-state machine or other formally defined automaton.
- *Logical inference*: Logical inference is a method of designing software that specifies preconditions and postconditions around each significant design block. Using mathematical logic, it develops the proof that those preconditions and postconditions must hold under all inputs. This allows the software engineer to predict software behavior without having to execute the software. Some integrated development environments (IDEs) include ways to represent these proofs and the design or code.

4.3. Prototyping Methods [1*, c12s2, 3*, c2s3.1, 6*, c7s3p5]

Software prototyping is an activity that generally creates incomplete or minimally functional versions of a software application, usually for trying out specific new features; soliciting feedback on software requirements or user interfaces; further exploring software requirements, software design, or implementation options; or gaining some other useful insight into the software. The software engineer selects a prototyping method to first understand the least understood software aspects or components. This approach contrasts with other software engineering methods that usually begin development with the best-understood portions first. Typically, the prototype does not become

the final software product without extensive development rework or refactoring.

This section briefly discusses prototyping styles, targets and evaluation techniques .

- *Prototyping style:* Prototyping styles describe the various approaches to developing prototypes. A prototype can be developed as throwaway code or a paper product , as an evolution of a working design, or as an executable specification. Different prototyping life cycle processes are typically used for each style. The style chosen is based on the type of results the project needs, the quality of the results needed and the results' urgency .
- *Prototyping target:* The prototyping target is the specific product served by the prototyping effort. Examples of prototyping targets are a requirements specification, an architectural design element or component, an algorithm, and a human-machine user interface.
- *Prototyping evaluation techniques:* The software engineer or other project stakeholders may use or evaluate the prototype in many ways , driven primarily by the underlying reasons that led to prototype development . Prototypes may be evaluated or tested against the implemented software or target requirements (e.g., a requirements prototype). The prototype might also serve as a model for future software development (e.g., as in a user interface specification).

4.4. Agile Methods [3*, c3, 6*, c7s3p7, 7*, c6, App. A, 16, 17]

Agile methods were developed in the 1990s to reduce the apparent large overhead associated with heavyweight, plan-based methods used in large-scale software development projects. Agile methods are

considered lightweight because of their short, iterative development cycles, self-organizing teams, simpler designs, code refactoring, test-driven development, frequent customer involvement and emphasis on creating a demonstrable working product with each development cycle.

Many Agile methods are available in the literature . Some more popular approaches, discussed here briefly, include rapid application development (RAD), eXtreme Programming (XP), Scrum, and feature-driven development (FDD).

- *RAD:* RAD methods are used primarily in data-intensive, business systems application development. RAD is enabled by special-purpose database development tools used by software engineers to quickly develop, test and deploy new or modified business applications.
- *XP:* This approach uses stories or scenarios for requirements, develops tests first, has direct customer involvement on the team (typically defining acceptance tests), uses pair programming, and provides continuous code refactoring and integration. Stories are decomposed into tasks, prioritized, estimated, developed and tested. Each software increment is tested with automated and manual tests. An increment may be released frequently, such as every couple of weeks .
- *Scrum:* This Agile approach is more project management-friendly than the others. The Scrum master manages the activities within the project increment . Each increment is called a *sprint* and lasts no more than 30 days. A product backlog item (PBI) list is developed; tasks from this list are identified, defined, prioritized and estimated. A

working version of the software is tested and released in each increment. Daily Scrum meetings ensure work is managed to the plan.

- *FDD*: This is a model-driven, short, iterative software development approach using a five-phase process: (1) develop a product model to scope the breadth of the domain, (2) create the list of needs or features, (3) build the feature development plan, (4) develop designs for iteration-specific features, and (5) code, test, and then integrate the features. FDD is similar to an incremental software development approach . It is similar to XP, except that code ownership is assigned to individuals rather than to the team. In addition, FDD emphasizes an overall architectural approach to the software, which promotes building features correctly the first time rather than rely on continual refactoring.

There are many more variations of Agile methods in the literature and in practice. There will always be a place for heavyweight, plan-based software engineering methods as well as places where Agile methods shine. In addition, new methods are arising from combinations of Agile and plan-based methods: Practitioners are defining these new methods to balance features from heavyweight and lightweight methods based primarily on organizational business needs. These business needs, as identified by project stakeholders, should and typically do drive the choice of software engineering method .

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	Budg en 2003 [1*]	Mellor and Balcer 2002 [2*]	Som merv ille 2011 [3*]	Pag e- Jon es 199 9 [4*]	Wi ng 199 0 [5*]	Bro oks hea r 200 8 [6*]]{B roo ksh ear , 200 8, Co mp ute r Sci enc e: An Ov erv iew }	Boe hm and Tu rne r 200 3 [7*]]
1. Modeling							
<i>1.1. Modeling Principles</i>	c2s2, c5s1, c5s2	c2s2	c5s0				
<i>1.2. Properties and Expression of Models</i>	c5s2, c5s3		c4s1.1p7, c4s6p3, c5s0p3				
<i>1.3. Syntax, Semantics and Pragmatics</i>		c2s2.2p6	c5s0				
<i>1.4. Preconditions, Postconditions and Invariants</i>		c4s4		c10s4p2, c10s5p2p 4			
2. Types of Models							
<i>2.1. Information Modeling</i>	c7s2.2		c8s1				

2.2. Behavioral Modeling	c7s2.1, c7s2.3, c7s2.4	c9s2	c5s4				
2.3. Structure Modeling	c7s2.5, c7s3.1, c7s3.2		c5s3	c4			
3. Analysis of Models							
<i>3.1. Analyzing for Completeness</i>			c4s1.1p7, c4s6		pp8-11		
<i>3.2. Analyzing for Consistency</i>			c4s1.1p7, c4s6		pp8-11		
<i>3.3. Analyzing for Correctness</i>					pp8-11		
<i>3.4. Traceability</i>			c4s7.1, c4s7.2				
<i>3.5. Interaction Analysis</i>		c10, c11	c29s1.1, c29s5	c5			
4. Software Engineering Methods							
<i>4.1. Heuristic Methods</i>	c13, c15, c16		c2s2.2, c7s1, c5s4.1				
<i>4.2. Formal Methods</i>	c18		c27		pp8-24		
<i>4.3. Prototyping Methods</i>	c12s2		c2s3.1			c7s3p5	
<i>4.4. Agile Methods</i>			c3			c7s3p7	c6, app. A

REFERENCES

- [1*] D. Budgen, *Software Design*, 2nd ed. New York: Addison-Wesley, 2003.
- [2*] S. J. Mellor and M. J. Balcer, *Executable UML: A Foundation for Model-Driven Architecture*, 1st ed. Boston: Addison-Wesley, 2002.
- [3*] I. Sommerville, *Software Engineering*, 9th ed. New York: Addison-Wesley, 2011.
- [4*] M. Page-Jones, *Fundamentals of Object-Oriented Design in UML*, 1st ed. Reading, MA: Addison-Wesley, 1999.
- [5*] J. M. Wing, “‘A Specifier’ s Introduction to Formal Methods,” *Computer*, vol. 23, pp. 8, 10-23, 1990.
- [6*] J. G. Brookshear, *Computer Science: An Overview*, 10th ed. Boston: Addison-Wesley, 2008.
- [7*] B. Boehm and R. Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston: Addison-Wesley, 2003.

CHAPTER 12

SOFTWARE QUALITY

ABBREVIATIONS

CI/CD	Continuous Integration/Continuous Delivery
CoSQ	Cost of Software Quality
COTS	Commercial Off-The-Shelf
FMEA	Failure Mode and Effects Analysis
FTA	Fault Tree Analysis
IV&V	Independent Verification and Validation
PDCA	Plan-Do-Check-Act
PSP	Personal Software Process
QFD	Quality Function Deployment
RCA	Root Cause Analysis
SCM	Software Configuration Management
SQA	Software Quality Assurance
SQAP	Software Quality Assurance Plan
SQC	Software Quality Control
SQM	Software Quality Management
VSEs	Very Small Entities
V&V	Verification and Validation

INTRODUCTION

What is software quality, and why is it so important that it is included in many knowledge areas (KAs) of the *SWEBOk Guide*? One reason is that the term *software quality* is overloaded. Software quality may refer to the desirable characteristics of software products, to the extent to which a particular software product has those characteristics (software product quality), and to the processes, tools and techniques used to achieve those characteristics (software process quality). Over the years,

authors and organizations have defined the term *quality* differently. Phil Crosby defined quality as “conformance to requirements” [2]. Watts Humphrey referred to it as “achieving excellent levels of “fitness for use” [3]. Meanwhile, IBM coined the phrase “market-driven quality,” where the “customer is the final arbiter” [4, p. 8].

More recently, software (product) quality has been defined as the “capability of a software product to satisfy stated and implied needs under specified conditions” [4] and as “the degree to which a software product meets established requirements; however, quality depends upon the degree to which those established requirements accurately represent stakeholder needs, wants, and expectations” [6]. Both definitions embrace the premise of conformance to requirements. Neither refers to different types of requirements (requirements categorized according to functionality, reliability, performance, dependability, or any other characteristic). Significantly, however, these definitions emphasize that quality is an important characteristic of requirements.

These definitions also illustrate another reason for the recurring discussions about software quality throughout the *SWEBOk Guide* — the often-unclear distinction between *software quality* and *software quality requirements* (“the *-ilities*” is a common shorthand for these terms). Software quality requirements (Quality of Service Constraints in the Software Requirements KA) are attributes of (or constraints on) functional requirements (what the system does). Software requirements may also specify resource use, a communication

protocol, or many other characteristics (Technology Constraints in the Software Requirements KA). This KA attempts to clarify by using *software quality* in the broadest sense from the definitions above and by using *software quality requirements* as constraints on functional requirements. Software product quality is achieved by conforming to all requirements regardless of specified characteristics or grouping or naming of requirements.

Software quality is also discussed in many other SWEBOK Knowledge Areas because it is a basic parameter of a software engineering effort. The primary goal for all engineered products is to deliver maximum stakeholder value while balancing the constraints of development and maintenance costs and schedule, sometimes characterized as *fitness for use*. Stakeholder value is expressed in requirements. For software products, stakeholders could value price (what they pay for the product), lead time (how fast they get the product), and software quality. (See the Software Requirements KA for a broader discussion of this.)

The software process quality aspect, which is implied by the above, must be made explicit. The quality of a software process can be also observed in process characteristics such as efficiency, effectiveness, usability, and learnability. Defects in that process will

likely show up as defects in the resulting software product, as well.

Finally, the Agile and DevOps movements aim at improving the software process and product quality through compliance by promoting quick iteration feedback loops and eliminating organizational silos by collocating users and software engineers. Other practices like pair programming and the automation of development, testing, and operations services also bring value, improve efficiency, and can detect defects early. (Refer to the Process KA for Agile life cycles and the Software Operations KA for more information on DevOps processes.)

This KA provides an overview of practices, tools, and techniques for understanding software quality and planning and appraising the state of software quality during development, maintenance and operation, from both a software product perspective and a software process perspective. Cited references provide additional details.

BREAKDOWN OF TOPICS FOR SOFTWARE QUALITY

The breakdown of topics for the Software Quality KA is presented in Figure 1.

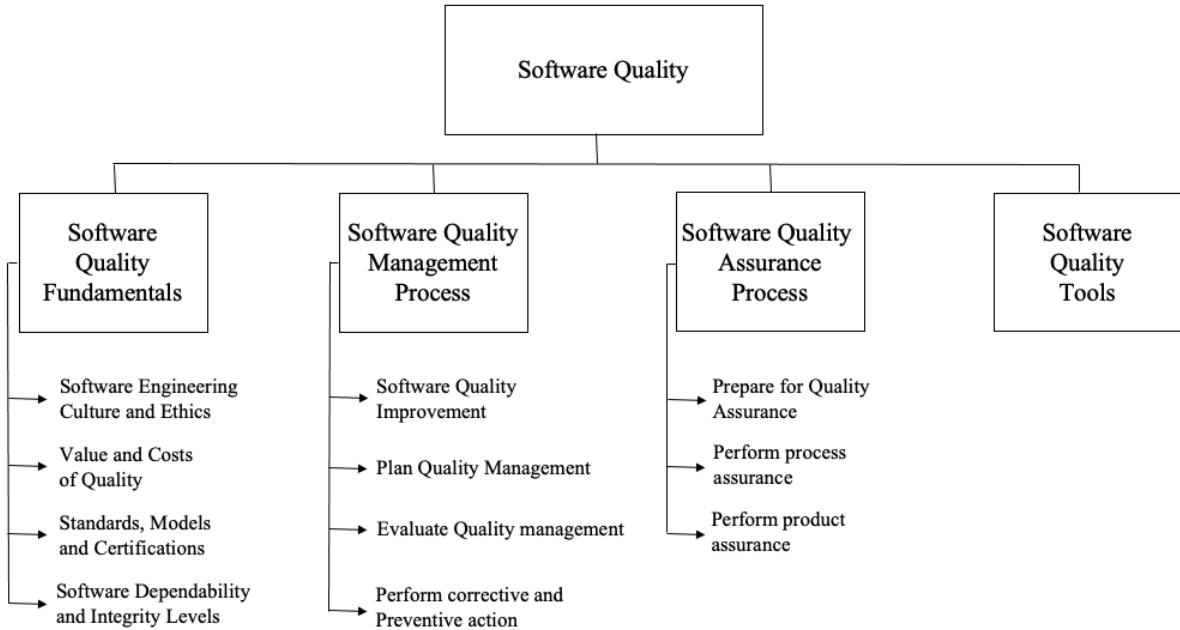


Figure 1. Breakdown of Topics for Software Quality

1. Software Quality Fundamentals

Agreeing on what constitutes software quality for all stakeholders and communicating that agreement to software engineers requires that the many aspects of quality be formally defined and communicated. The main challenges the software engineer faces to ensure quality include the following:

- Difficulty in clearly defining requirements;
- Maintaining effective communication with the client/user;
- Deviations from specifications;
- Architecture and design errors;
- Coding errors;
- Noncompliance with current processes/procedures;
- Inadequate work product reviews and tests;
- Documentation errors.

Software quality is defined as “conformance to established requirements; the capability of

a software product to satisfy stated and implied needs when under specified conditions” [6]. It is further defined “by the degree to which a software product meets established requirements; however, quality depends upon the degree to which those established requirements accurately represent stakeholder needs, wants, and expectations” [6]. Quality often means the absence of defects. The word *defect* is overloaded with too many meanings, as engineers and others use the word to refer to all different types of anomalies. However, different engineering cultures and standards often understand “defect” and other terms as having more specific meanings. To avoid confusion, software engineers should use the meaning provided by their standards [14]:

- *Error*: “A human action that produces an incorrect result.” Also called *human error*;
- *Defect*: (*synonym of a fault*) An “imperfection or deficiency in a work product where that work product does

- not meet its requirements or specifications and needs to be either repaired or replaced.” A defect is inserted when a person developing the software makes an error. It hides in the software until (and if) it is discovered;
- *Failure*: The “termination of the ability of a system to perform a required function or its inability to perform within previously specified limits; an externally visible deviation from the system’s specification event in which a system or system component does not perform a required function within specified limits.” A failure is produced when the software executes a defect.

A software engineer should understand software quality concepts, characteristics, and values and their application to the many development, maintenance, and operation activities. An important concept is that the software requirements are expected to define the required software quality attributes. Furthermore, software requirements influence the measurement methods and acceptance criteria for assessing how the software and related work products achieve the desired quality levels. Another important concept is that software quality should be planned early and assessed at many milestones during the software life cycle. Finally, how to adapt software quality assurance (SQA) activities to accommodate Agile software development is presented in detail in the Institute of Electrical and Electronics Engineers (IEEE) Standard 730:2014 [6].

1.1. Software Engineering Culture and Ethics [1, c1s1.6; c2] [5*]*

An organization’s culture affects how software engineers influence software quality. As Iberle explains [19], software engineering practices vary depending on the

business model (e.g., custom, mass-market, commercial, firmware) and the industry where the software engineers work. Software engineers are expected to share a commitment to software quality in the context of their industry and as part of their culture. A healthy software engineering culture includes many characteristics, such as the understanding that trade-offs among cost, schedule and quality are a basic tenet of any product’s engineering. A strong software engineering ethic assumes that engineers accurately report information, conditions and outcomes related to quality. Ethics also play a significant role in software quality in the professional culture of engineering, and in the attitudes of software engineers. The IEEE Computer Society and the Association for Computing Machinery (ACM) have developed a code of ethics and professional practice. (See Codes of Ethics and Professional Conduct in the Software Engineering Professional Practice KA.)

1.2. Value and Costs of Quality [1, c2s2.2]*

One major factor driving resistance to implementing SQA is its perceived high cost. However, not implementing basic SQA activities can be costly as well. Software engineers should inform their administration of the risks they take when they are not fully committed to quality. This can be done by explaining the cost of software quality concepts to management. Cost of software quality (CoSQ) is defined as the sum of the following project costs:

- Cost of planning and construction activities (e.g., planning, designing, development);
- Cost of prevention activities (process improvement, tools, training);
- Cost of appraisal activities for defect detection (e.g., reviews, audits, testing);

- Cost of nonconformance rework (internal failure cost and external failure cost).

The CoSQ can be broken down into two top-level categories: *conformance cost* and *nonconformance cost*. Conformance cost is the total of all investments in error and defect detection (*appraisal*) and prevention activities. Appraisal costs arise from project activities that are intended to find errors and defects. These include testing (as detailed in the Software Testing KA) and reviews and audits (as detailed later in this KA). Appraisal costs extend to subcontracted software suppliers, if any. Prevention costs include investments in software process improvement (SPI) efforts, quality infrastructure, quality tools, work product templates and training. These costs might not be specific to a project; they often span the larger organization.

Nonconformance cost is the total of all spending dealing with errors and defects that have been detected. Pre-delivery costs are those incurred to repair errors and defects found during appraisal activities and discovered before the software product is delivered to the customer. Post-Delivery costs include those incurred responding to software failures discovered after delivery to the customer. External costs include the rework needed to repair and test an updated release. However, the financial impact on the customer who encounters a failure is just as important. For example, the customer's lost productivity, lost data, and potential loss of reputation in the marketplace must be acknowledged and accounted for. Beyond the impact on the customer, low-quality software can also impact the public and the environment. Software engineers should seek the optimal CoSQ — the minimal total cost for a specified quality level.

1.3. Standards, Models and Certifications [1*, c4] [7, c24s24.2]

Sound use of software engineering software standards and software process assessment and improvement improves software quality. One of the key general software engineering standards is ISO/IEC/IEEE 12207:2017, which describes the software life cycle processes. Foremost, software engineers should know the key software engineering standards that apply to their specific industry. As Iberle discussed [19], the practices software engineers use vary greatly depending on the industry, business model and organizational culture where they work. For example, IEEE 1228:1994 Standard for Software Safety Plans and IEEE 1633:2016 Recommended Practice on Software Reliability target industries where safety and reliability are important.

The PDCA paradigms differ from standards in that it often proposes “best practices” for software engineers from a specific perspective. (Refer to the Software Engineering Process KA for more information about the PDCA paradigm for software.)

Other industry “best practices” models such as the Control Objectives for Information and Related Technologies (COBIT) for information technology governance, the Project Management Body of Knowledge (PMBOK®) for project management, the Business Analysis Body of Knowledge (BABOK®), the Capability Maturity Model Integration (CMMI) and The Open Group Architecture Framework (TOGAF) propose software related practices that can improve the quality of software processes and products. Software organizations can also consider the possible advantages of obtaining registrations or certifications (e.g., ISO 9001 for quality, ISO 27001 for security, ISO 20000 for operations), and software engineers can also obtain Scrum and Scaled

Agile Framework® (SAFe®) certifications for Agile processes. The use of these models and certifications have been shown to augment stakeholders' confidence that the software engineers' knowledge and skills are up to date and recognized internationally.

1.4. *Software Dependability and Integrity Levels* [1*, c4s4.8.2, c7s7.3.3] [11]

Software-intensive and safety-critical systems are those in which a system failure could harm human life, other living things, physical structures, or the environment. The software in these systems is considered safety-critical and requires the use of systematic methods and tools to ensure its high level of quality. A growing number of industries are using increasing numbers of safety-critical software, including transportation systems, chemical and nuclear plants, and medical devices. Software failure in these systems could have catastrophic effects. Engineers use industry standards, such as software considerations in airborne systems and equipment certification DO-178C [8] and railway applications EN 50128 [18], and emerging processes, tools, and techniques to develop safety-critical software more safely. These standards, tools and techniques reduce the risk of injecting faults into the software and thus improve software availability, reliability, and maintainability. Software engineers and their managers must understand the threats and issues and develop the skills needed to anticipate and prevent accidents before they occur [15].

Safety-critical software can be categorized as *direct* or *indirect*. Direct software is embedded in a safety-critical system, such as an aircraft's flight control computer. Indirect software includes software applications used to develop safety-critical software. Indirect software is included in software engineering

environments and software test environments.

Three complementary techniques for reducing failure risk are avoidance, detection and removal, and damage limitation. These techniques impact software functional requirements, performance requirements and development processes. Increasing risk implies increasing SQA and more rigorous review techniques such as inspections [16]. Higher risk levels might necessitate more thorough inspections of requirements, design, and code, or the use of more formal verification and validation techniques. Another technique for managing and controlling software risk is building assurance cases. An *assurance case* is a reasoned, auditable artifact created to support the contention that its claim or claims are satisfied. It contains the following relationships: one or more claims about properties, arguments that logically link the evidence and any assumptions to the claims, and a body of evidence and assumptions supporting these arguments [9].

1.4.1. *Dependability* [7, c10]

In cases where system failure may have severe consequences, overall dependability (e.g., hardware, software, and human or operational dependability) is the main quality requirement, aside from basic software functionality, for the following reasons: System failures affect many people; users often reject systems that are unreliable, unsafe, or insecure; system failure costs could be enormous; and undependable systems might cause information loss. Many standards address different perspectives of dependability, such as reliability and availability. System and software dependability regroups several related quality characteristics: availability, reliability, maintainability and supportability, risk assessment, and safety

and security [21]. When developing dependable software, engineers can apply tools and techniques to reduce the risk of injecting faults into the intermediate deliverables or the final software product. They can use static, dynamic, or formal methods for verification and validation (V&V), and testing processes, as well as other specialized techniques, methods, and tools to identify defects that affect dependability as early as possible in the software life cycle [7*, c10.5]. Additionally, they may have to incorporate specific mechanisms into the software to guard against external attacks and to tolerate faults during its operation.

1.4.2. Integrity Levels of Software [1*, c4s4.8.2, c7s7.3.3] [11]

Defining integrity levels is a method of risk management. An *integrity level* is “a value representing project-unique characteristics (e.g., complexity, criticality, risk, safety level, security level, desired performance, and reliability) that define the importance of the system, software, or hardware to the user” [11]. The characteristics used to determine software integrity level vary depending on the intended application and use of the system. The software is a part of the system, and its integrity level is determined as a part of that system.

The assigned software integrity levels might change as the software evolves. Design, coding, procedural and technology features implemented in the system or software can raise or lower the assigned software integrity levels. The software integrity levels established for a project result from agreements among the acquirer, supplier, developer, and independent assurance authorities. A *software integrity level scheme* is used to determine software integrity levels [11].

Software engineers should know that in certain safety-critical industries, such as avionics, railways, nuclear power, medical devices and many others, industry-specific guidance can require a certain level of independence for software quality activities and can assign minimum V&V techniques to be used by integrity level (example of such techniques are: usability analysis, algorithm analysis, boundary value analysis, data flow analysis, walk-through review [11][26]).

2. Software Quality Management Process

“Software quality management” (SQM) is concerned coordinating activities to direct and control an organization with regard to software quality” [6]. The purpose of the Quality Management process is to assure that products, services, and implementations of the quality management process meet organizational and project quality objectives and achieve customer satisfaction.

Software engineers can learn about the SQM process in the many software engineering standards, models, and certifications available and used widely in the industry.

An important concept of SQM is the design and upkeep of a Quality Management System (QMS). As proposed by ISO 9001, a QMS defines processes, process owners, requirements for the processes, measurements of the processes and their outputs, and feedback channels throughout the whole software life cycle. A QMS comprises many key activities: SQA, V&V, reviews and audits, software configuration management (SCM), and requires policies, procedures, and processes to ensure that everyone involved understands what is expected in terms of software process and product quality. For a QMS to be effective, management support is imperative. Management support implies that projects are trained to the QMS requirement and have

enough resources to achieve the quality goal defined for it. Management sponsorship should be solicited frequently during software project review to ensure software quality activities are executed and nonconformities addressed.

For a software project, software quality processes consist of tasks and techniques to indicate how software plans (e.g., software management, development, quality management or configuration management plans) are implemented and how well the intermediate and final products meet their specified requirements. Results from these tasks are assembled in reports for management. SQM process management is tasked with ensuring that the report results are accurate and acted upon.

Risk management can also play an important role in delivering quality software. Incorporating disciplined risk analysis and management techniques into the software life cycle processes can help improve product quality. (See the Software Engineering Management KA for related material on risk management.)

2.1. Software Quality Improvement [1, s9.9 and c9] [2] [3]*

Software Quality Improvement (SQI) is done using many different approaches within the software industry, including software process improvement (SPI), Six Sigma, Lean, and Kaizen just to name a few. For example, the SPI activities seek to improve process effectiveness, efficiency, and other characteristics to improve software quality. For example, although SPI could be included in any of the first three categories, many organizations organize SPI into a separate category that might span many projects.

Software product quality can be improved using Lean principles as well as an iterative process of continual improvement, which

includes management control, coordination of activities, and feedback from many concurrent processes: (1) the process of improving the software life cycle processes; (2) the process of fault/defect categorization, detection, removal, and prevention; and (3) a personal improvement process.

The theory and concepts behind quality improvement — such as building quality through the prevention and early detection of defects, continual improvement, and stakeholder focus — are also pertinent to software engineering. These concepts are based on the work of experts in quality who have stated that a product's quality is directly linked to the quality of the process used to create it. Improvement models such as the Plan-Do-Check-Act (PDCA) improvement cycle, evolutionary delivery, Kaizen, and techniques like quality function deployment (QFD) offer ways to specify quality objectives and determine whether they are met.

Finally, since software engineering is a complex process, it cannot be reduced to a cookbook of procedures. To complement the process and tools improvement movement, Humphrey proposed the personal software process (PSP) for software engineers to also assess their skills and knowledge constantly and continually improve them as well.

2.2. Plan Quality Management

Software quality planning includes determining which quality standards and models are to be used, defining specific quality goals, estimating the effort to be used to achieve each goal; and deciding at what milestone the software quality activity should take place. In some cases, software quality planning also includes defining the software quality processes to be used.

First, the software organization must commit to quality by establishing their Quality Management System (QMS) which includes quality management policies, objectives, and procedures. This requires that the responsibility and authority for implementing the QMS are assigned and that they are independent of current project management teams.

An approved organizational policy, about software quality, help in guiding projects and products development decisions as well as behavior of personnel. Software engineers should promote the use of graphically represented processes and procedures that implement the quality policy and explain the roles, activities to be executed and the expected results of key software engineering activities. Consequently, for a QMS to be used in improvement its processes should be documented with its user in mind and identify where quality controls are to be verified. Finally, procedures explain in detail what steps are taken to execute a specific activity.

2.3. Evaluate Quality Management

Once the QMS is in place, the ISO/IEC Technical Specification TS 33061:2021 [22] Standard defines a process assessment model for software life cycle processes using five process capabilities levels (from level 0: incomplete to level 5: optimizing process). Additionally, software engineers can assess the maturity of their QMS activities in their software projects using the IEEE 730:2014 Standard guidance [6]. Management sponsorship supports process and product evaluations and the resulting findings feed into an improvement program is developed identifying detailed actions and improvement projects to be addressed in a feasible time frame. Periodically, the software engineers will gather and analyze quality assurance evaluation results. This can be achieved by

looking at quality measures and defect characterization produced by the projects.

2.3.1. Software Quality Measurement [1, c10] [7, c24s24.5]*

Software quality measurements are used to support decision-making. With the increasing sophistication of software, quality questions go beyond whether the software works to how well it achieves measurable quality goals. Quantifying some attribute of software can help engineers evaluate its quality or the quality of its process. (Process measurement is described in detail in the Process KA.)

Software quality measurement helps engineers make determinations about software quality (because models of software product quality include measures to determine the degree to which the software product achieves quality goals); managerial questions about effort, cost, and schedule; when to stop testing and release a product (see Test-Related Measures in the Software Testing KA); and the efficacy of process improvement efforts.

The CoSQ assurance activities are an issue frequently raised in deciding how a project or a software development and maintenance organization should be organized. Often, generic models of cost are used; these models are based on when a defect is found and how much effort it takes to fix the defect relative to finding the defect earlier in development. Software quality measurement data collected internally may offer a better picture of cost within the project or organization. Although the software quality measurement data may be useful by itself (e.g., the number of defective requirements or the proportion of defective requirements), mathematical and graphical techniques can help project stakeholders interpret the measures. (See the Engineering Mathematical Foundations KA.) These techniques include the following:

- Descriptive statistics-based analysis (e.g., Pareto analysis, run charts, scatter plots, normal distribution);
- Statistical tests (e.g., the binomial test, chi-squared test);
- Trend analysis (e.g., control charts; see *The Quality Toolbox* in Further Reading);
- Prediction (e.g., reliability models).

Descriptive statistics-based techniques and tests often provide a snapshot of the more troublesome areas of the software product under examination. The resulting charts and graphs are visualization aids decision-makers can use to focus resources and conduct process improvements where they seem most needed. Results from trend analysis may indicate that a schedule is being or that certain classes of faults may become more likely unless some corrective action is taken in development. The predictive techniques help estimate testing effort and schedule and predict failures. (More discussion on measurement in general appears in the Software Engineering Process and Software Engineering Management KAs. More specific information on testing measurement is presented in the Software Testing KA.) Software quality measurement also includes measuring defect occurrences and applying statistical methods to understand what types of defects occur most frequently. Three widely used software quality measurements are error density (number of errors per unit size of documents/software), defect density (number of defects found divided by the size of the software), and failure rate (mean time to failure). Reliability models are built from failure data collected during software testing or from software in service and thus can be used to estimate the probability of future failures and assist in decisions about when to stop testing. This information can be used in SPI to determine methods to prevent, reduce or eliminate defect recurrence. The

information also helps engineers understand trends, how well detection and containment techniques are working, and how well the development and maintenance processes are progressing. They can use these measurement methods to develop defect profiles for a specific application domain. Then, for the next software project within that organization, the profiles can be used to guide the SQM processes — that is, to focus effort on where problems are most likely to occur. Similarly, benchmarks, or defect counts typical of that domain, may help engineers determine when the product is ready for delivery. (Discussion about using measurement data to improve development and maintenance processes appears in the Software Engineering Management and Software Engineering Process KAs.)

2.4. Perform Corrective and Preventive Actions

It is important that when quality management objectives are not met, corrective actions be documented and submitted so that the QMS be improved to prevent problem from reoccurring in future software projects. This requires that project participants have a way of reporting software engineering process and tools problems to an independent organization that will document and monitor the progress of the corrective actions and inform the relevant stakeholders.

2.4.1. Defect Characterization [1*, c1s3]

To help in the elimination of the cause or causes of an existing nonconformity or undesirable situation to prevent recurrence, software engineers can use Software Quality Control (SQC) techniques to find errors, defects, and failures in their processes and products. When tracking errors, defects and failures, the software engineer is interested in the number and types of incidents. Numbers alone, without classification, might be insufficient to help in identifying the

underlying causes and thus to prevent them in the future. Therefore, software engineers should establish a meaningful defect classification taxonomy to describe and categorize such anomalies. One probable action resulting from peer reviews and testing findings is to remove these errors and defects early from the work product under examination.

Other SQM activities attempt to eliminate their causes (e.g., root cause analysis (RCA)). RCA activities include analyzing and summarizing the findings to find root causes and using measurement techniques to improve the software engineering processes, techniques and tools. (Process improvement is primarily discussed in the Software Engineering Process KA. RCA is further discussed in the Engineering Mathematical Foundations KA.)

Data on errors and defects found during SQA and control techniques may be lost unless they are recorded. For some techniques (e.g., peer reviews and inspections), software engineers are present to record such data and to address issues and make decisions. In addition, when automated tools are used (see Topic 4, Software Quality Tools), the tool output may provide defect trends reports that can be provided to the organization's management.

3. Software Quality Assurance Process

3.1. *Prepare for Quality Assurance [1*, c1s1.5] [6]*

Software quality assurance (SQA) is defined as “a set of activities that define and assess the adequacy of software processes to provide evidence that establishes confidence that the software processes are appropriate for and produce software products of suitable quality for their intended purposes.” To correct a common misunderstanding, SQA is

not only testing of a software. A key attribute of SQA, in critical systems, is the objectivity of the SQA function concerning the quality of a software product. In this case, the SQA function might also be organizationally independent of the project; that is, free from technical, managerial, and financial pressures [6]. SQA has two aspects: product assurance and process assurance, which are explained in Section 2.3.

The software quality plan (in some industry sectors, it is termed the software quality assurance plan (SQAP)) defines the activities and tasks used to ensure that software developed for a specific product satisfies the project's established requirements and user needs within project cost and schedule constraints and is commensurate with project risks. The SQAP first ensures that quality targets are clearly defined and understood.

The SQAP's quality activities and tasks are specified, along with their costs, resource requirements, objectives, and schedule in relation to related objectives, in the software engineering management, software development and software maintenance plans. The SQAP identifies documents, standards, practices, and conventions governing the project and how these items are checked and monitored to ensure adequacy and compliance. The SQAP also identifies measures; statistical techniques; procedures for problem reporting and corrective action; resources such as tools, techniques, and methodologies; security for physical media; training; and SQA reporting and documentation. Moreover, the SQAP addresses the SQA activities of any other type of activity described in the software plans — such as procurement of supplier software for the project, commercial off-the-shelf (COTS) software installation and service after software delivery. It can also contain acceptance criteria and reporting and management activities that are critical to software quality. The SQA plan should not

conflict with the software configuration management plan (see the Software Configuration Management KA) or any other relevant project planning artifact.

Software quality encompasses several perspectives: the software process quality, the software end-product quality and the software work products (also called *intermediary products*) quality. The next sections cover each perspective of software quality knowledge a software engineer must have.

3.2. *Perform Process Assurance* [1*, c3s3.2–s3.3; ; c8 ;c9] [7, c25]

Crosby [2] and Humphrey [3] have demonstrated that software quality management (SQM) and software engineering process quality have a direct effect on the quality of the final software product. (Models and criteria that evaluate and improve the capabilities of software organizations are primarily project organization and management considerations and, as such, are covered in the Software Engineering Management and Software Engineering Process KAs.) International Organization for Standardization (ISO) 9000 [10] proposes another process quality perspective, where a management system that oversees the processes' actors, activities, controls, input, and outputs ensures the quality of outputs (e.g., work products and final product). A *management system* is defined as a “set of interrelated or interacting elements of an organization to establish policies and objectives, and processes to achieve those objectives” [10]. This perspective requires software engineering organizations to take the time to describe their policies, processes, and procedures with enough detail that software engineer roles and responsibilities are clear during life cycle activities (as detailed in the Software Engineering Process KA).

SQA activities, listed in the IEEE 730:2014

Standard [6], describe the many quality assurance activities that should be conducted early in a software project’s life cycle to ensure quality. Software engineers should be aware of the need to plan and execute SQA activities at certain project milestones and keep records of their execution. These activities consist of document and code reviews as well as verification and validation (V&V) activities, including testing (as detailed in Section 3.3.1 of this KA), which evaluate the output of a process’s compliance with its requirements and specifications.

Finally, software configuration management (SCM) is an important activity to ensure the quality of work products and software. *Configuration management* is defined as the “discipline applying technical and administrative direction and surveillance to:

- identify and document the functional and physical characteristics of a configuration item;
- control changes to those characteristics;
- record and report change processing and implementation status;
- verify compliance with specified requirements.”

Software engineers should identify which work products and software artifacts require configuration management. In addition, they should be familiar with source code versioning processes, which involve keeping track of baselined and incremental versions of the software and ensuring that changes different developers make do not interfere with one another, and they should know how to operate the version control tool kit. (Refer to the Software Configuration Management KA for more information about this process.)

3.3. *Perform Product Assurance* [1*, s3.2–s3.3] [7, c4s4.1.2]

First, the software engineer must determine the real purpose of the software to be

designed and constructed. Stakeholder requirements are paramount here. They include quality requirements (called *Quality of Service Constraints* in the Software Requirements KA) and functional requirements. Thus, software engineers are responsible for eliciting quality requirements that might not be explicit at the outset and for understanding their importance and the difficulty in defining them, measuring them, and establishing them for final acceptance. Software engineers should understand how to define quality requirements as well as their quality targets to ensure they can effectively be measured at the acceptance stage of the project. During the project planning, software engineers must keep these quality requirements in mind. They must also anticipate potential additional development costs if attributes such as safety, security and dependability are important.

An international standard on what constitutes a software product's many measurable quality characteristics was reached and is described in ISO/IEC 25010:2011 [4]. This standard proposes several software product quality models, consisting of characteristics and sub-characteristics, for software product quality and software quality in use. Another is IEEE 982.1:2005 Standard Dictionary of Measures to Produce Reliable Software. These software characteristics are commonly called *product quality requirements*, which are nonfunctional software requirements [7* C4s4.1.2]. Software engineers should know the many software characteristics that can be planned, implemented, and measured during software construction (e.g., functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability). Software engineers should also know that certain quality characteristics have conflicting impacts. For example, trying to augment the security characteristic by encrypting data might adversely affect the performance

characteristic. This international standard also proposes a general data quality model that focuses on data quality as part of a computer system and defines quality characteristics for target data used by humans and systems.

Another software product quality perspective is the quality of work products. The term *work product* means any artifact resulting from a process used to create the final software product. Work products include system/subsystem specifications, software requirements specifications for a system's software components, software design descriptions, source code, software test documentation and test reports. Sound engineering practice requires that intermediate work products relevant to quality be evaluated using work product reviews and inspections (discussed later in this chapter) throughout the software engineering process.

3.3.1. V&V and Testing [1*, c7] [11]

Verification ensures that the product is built correctly in that the output products of a life cycle phase meet the specifications imposed on them in previous phases. Verification is defined as “the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase” [11]. Alternatively, *validation* ensures that the right product is built — the product fulfills its specific intended purpose. It is defined as “the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.”

The purpose of V&V is to help the development organization build quality into the software throughout the development life cycle. V&V includes software testing tasks. Software testing is a necessary activity to ensure product quality. However, in most cases, software testing is insufficient to

establish confidence that the software fits its intended use. V&V tasks listed in IEEE Standard 1012:2016 [11] objectively assess products and processes throughout the life cycle. This assessment demonstrates whether the requirements are correct, complete, accurate, consistent, and testable. The verification process and the validation process should begin early in development or maintenance. This prevents defects late in the life cycle, which would incur rework and significantly increase costs. Software engineers should identify the product integrity level and ensure the minimum V&V tasks are assigned for key product features concerning both the product's immediate predecessor and the planned specifications. Optional V&V tasks are also listed and can improve software product quality. Keeping a record of the traceability among software work products can help augment the quality of the V&V activities. *Traceability* is defined as the “ability to trace the history, application or location of an object” [14].

Early planning of V&V activities ensures that each resource, role, and responsibility is clearly assigned. The resulting V&V plan documents the various resources and their roles and SQA activities, as well as the techniques and tools to be used. Software engineers should choose and apply the proper V&V task depending on the software integrity level. (Refer to Section 3.1.3.) V&V can also be executed by an independent organization for very critical software. Independent verification and validation (IV&V) are defined as “V&V performed by an organization that is technically, managerially, and financially independent of the development organization” [11].

Software V&V tasks can be sorted into static, dynamic and formal tasks [20]. Dynamic techniques involve executing the software; static techniques involve analyzing documents and source code but not executing the software; formal techniques use

mathematics and formal specification languages.

3.3.2. Static Analysis Techniques

Static analysis techniques directly analyze a work product’s content and structure (including requirements, interface specifications, designs, and models) without executing the software. Tools and techniques for statically examining software work can help software engineers in this task. For example, code reading, peer review of a work product, and static analysis of source code control flow are considered static techniques because they do not involve executing the software code.

3.3.3. Dynamic Analysis Techniques

Dynamic analysis techniques involve executing or simulating the software code, looking for errors and defects. Different dynamic techniques are performed throughout software development, maintenance, and operation. Generally, these are testing techniques, but simulation and model analysis are considered dynamic analysis techniques. (See the Software Engineering Models and Methods KA.) In addition, black box testing is considered a dynamic analysis technique, as the software engineer analyzes the output received following the entry of inputs. (See the Software Testing KA.)

3.3.4. Formal Analysis Techniques

Formal analysis techniques (also called *formal methods*) are “mathematical approaches to software development where you define a formal model of the software. You may then formally analyze this model to search for errors and inconsistencies” [7*, c10s10.5]. Sometimes, the software requirements may be written using a more formal specification language known as formal methods. They are notably used to verify software requirements and designs. They have mostly been used to verify crucial

parts of critical systems, such as specific security and safety requirements. (See also Formal Methods in the Software Engineering Models and Methods KA.) Different groups may perform testing during software development, including groups independent of the development team. The Software Testing KA is devoted entirely to this subject.

3.3.5. Software Quality Control and Testing [1*, c7s7.10]

Testing is considered an important product quality control activity part of a software development project's V&V processes. Quality Control is a set of activities that measure, evaluate and report on the quality of software project artifacts throughout the project life cycle [25]. Software testing is an important quality control activity to ensure software quality. Software testing is one of many verification activities that confirm that software development output meets input requirements. IEEE 730:2014 [6] lists the many testing and retesting activities software engineers should plan, execute, and record. It also recommends that testing completion criteria be set. Software engineers should plan the testing activities, including levels, techniques, measures, and tools. (Refer to the Testing KA for details about the knowledge software engineers should have about software testing.)

3.3.6. Technical Reviews and Audits [1*, c5, c6] [23, s4, s5]

We have seen SQC techniques for assessing the quality of the software in the previous section. For the other artefacts, product quality control is assessed using reviews and inspections of these work products. These SQC activities are planned and executed during development, maintenance, and operations activities [17]. Peer reviews are defined as “the review of work products performed by peers during development of

the work products to identify defects for removal” [14]. For example, during software development, a code review (often done by using a pull request technique/tool) occurs when a peer reviews the code, often at the software developer’s request, before it can be merged into a project.

Reviews are valuable because they can identify issues early in development or even before a component is designed. Fixing a defect in a component that has been coded is much more expensive than catching it beforehand. Review and audit processes are broadly defined as static activities, meaning that no software or models are executed. Instead, they examine software engineering artifacts (also called *intermediary* or *work products*) concerning standards established by the organization or project for those artifacts.

Different types of work product reviews (e.g., formal, and informal) are distinguished by purpose, level of independence, tools and techniques used, roles involved, and by the subject of the activity. Reviews play important roles in software quality, in SCM, and in the sharing of knowledge among colleagues. However, these different roles share a single purpose — to ensure the quality of the delivered products. Reviews should be part of the software engineering culture and should be planned, executed, and documented during the software life cycle. In Agile life cycles, pair programming invites continuous reviews. Different review types for work products are described in the ISO/IEC 20246:2017 Standard [12]: Ad hoc reviews — unstructured reviews where each reviewer is expected to find as many defects as possible of any type; Checklist-based reviews — systematic reviews identifying issues based on checklists; Scenario-based reviews — reviews where reviewers are provided with structured guidelines on how to read through the work product under review; Perspective-based reviews —

reviews where reviewers take on different stakeholder viewpoints and review the work product from that stakeholder's viewpoint; and Role-based reviews — reviews in which the reviewer evaluates the work product from the perspective of various stakeholder roles, which might differ from their daily role.

Audits are more formal activities that are often mandated to be performed by third parties to ensure independence. In mature organizations, technical reviews and audits are fully integrated with the overall project plans. Therefore, technical reviews and audits should be planned, approved, and conducted. Although a project audit often addresses the whole project's current state, technical reviews can also be more focused and address a specific project phase [24]. System requirements reviews help ensure that the level of understanding of top-level system requirements is adequate to support further requirements analysis and design activities and that the system can proceed into initial system design with acceptable risk; System functional or preliminary design reviews help ensure that the system under review can proceed into preliminary or detailed design with acceptable risk and that all system requirements and functional performance requirements derived from the approved preliminary system specification are defined and consistent with the project budget, program schedule, risk, and other program and system constraints; Preliminary design reviews help ensure that the preliminary design for the system under review is sufficiently mature and ready to proceed into detailed design and can meet the stated performance requirements within program budget, schedule, risk and other program and system constraints; Test readiness reviews assess test objectives, test methods and procedures, test scope, safety, readiness for the project test and evaluation, and whether test resources have been properly identified and obtained; Production

readiness reviews ascertain that the system design is ready for production and that the project has accomplished adequate production planning for entering production.

4. Software Quality Tools [1*, c3s3.2.3, c7s7.8.1, c7s7.11]

Software tools improve software quality. Simple tools can be forms and checklists (e.g., a requirements traceability matrix or a code review checklist). But automated tools can also be of great help to improve software efficiency and quality. Examples of automated tools are tools that allow code versioning/branching (e.g., Git) and pull requests for code review. DevOps tools in services/scripts like on-demand environments, continuous integration/continuous delivery (CI/CD), code quality assessment, and automated testing are important contributors to software quality. (See the Software Operations KA discussion about tools.)

These tools are known as static and dynamic analysis tools. Static analysis tools input source code, perform syntactical and semantic analysis without executing the code, and present results to users. There is a large variety in the depth, thoroughness and scope of static analysis tools that can be applied to artifacts, including models, and source code. (See the Software Construction, Software Testing, and Software Maintenance KAs for descriptions of dynamic analysis tools.) Categories of static analysis tools include the following: Tools that facilitate and partially automate reviews and inspections of documents and code. These tools can route work to different participants to partially automate and control the review process. In addition, they allow users to enter defects found during inspections and reviews for later removal; Tools that help organizations perform software safety hazard analysis. These tools provide, for example,

automated support for failure mode and effects analysis (FMEA) and fault tree analysis (FTA); Tools that support tracking of software problems. These tools enable entry of anomalies discovered during software testing and subsequent analysis, disposition, and resolution. Some tools include support for workflow and for tracking problem resolution status; and Tools that analyze data captured from software

engineering environments and software test environments and produce visual displays of quantified data in graphs, charts, and tables. These tools sometimes include the functionality to perform statistical analysis on data sets (to discern trends and make forecasts). Some of these tools provide defect and removal injection rates, defect densities, yields, and distribution of defect injection and removal for each life cycle phase.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

Topic	Laporte and April 2018 [1*]	Sommerville 2016 [7*]	IEEE Software Code of Ethics [5*]	Wiegers 2013 [13*]
1. Software Quality Fundamentals				
1.1. Software Engineering Culture and Ethics	Ch. 1s1.6, Ch. 2		X	
1.2. Value and Costs of Quality	Ch. 2s2.2			
1.3. Standards, Models and Certifications	Ch. 4	Ch. 24s24.2		
1.5. Software Dependability and Integrity Levels	Ch. 4s4.8.2 Ch. 7s7.3.3	Ch. 10		
2. Software Quality Management Process				
2.1. Software Quality Improvement	Ch 9s9.9			
2.2. Plan Quality Management				
2.3. Evaluate Quality Management	Ch. 10	Ch. 24s24.5		
2.4. Perform Corrective and Preventive action	Ch. 1,s3			
3. Software Quality Assurance Process				
3.1. Prepare for Quality Assurance	Ch. 1s1.5			Ch. 14
3.2. Perform Process Assurance	Ch. 3s3.2-s3.3 Ch. 8 Ch. 9	Ch. 25		

3.3. Perform product assurance	Ch. 3s3.2-3.3 Ch. 7 Ch. 5	Ch. 4s4.1.2		
4. Software Quality Tools	c3s3.2.3, c7s7.8.1, c7s7.11		x	

FURTHER READINGS

IEEE 730-2014, “IEEE Standard for Software Quality Assurance Processes,” 2014 [6]. Requirements for initiating, planning, controlling, and executing the Software Quality Assurance processes of a software development or maintenance project are established in this standard.

IEEE Std 1012-2016, “IEEE Standard for System, Software, and Hardware Verification and Validation,” 2016 [11]. Verification and validation (V&V) processes are used to determine whether the development products of a given activity conform to that activity’s requirements and whether the product satisfies its intended use and user needs. V&V life cycle process requirements are specified for different integrity levels.

ISO/IEC Std 20246-2017, “Software and Systems Engineering — Work Product Reviews,” 2017 [12]. This international standard establishes a generic framework for work product reviews that can be referenced

and used by all organizations involved in the management, development, testing and maintenance of systems and software.

N. Leveson, *Safeware: System Safety and Computers* [15]. This book describes the importance of software safety practices and how these practices can be incorporated into software development projects.

T. Gilb and D. Graham, *Software Inspection* [16]. This book introduces measurement and statistical sampling for reviews and defects. It presents techniques that produce quantified results for reducing defects, improving productivity, tracking projects and creating documentation.

K. E. Wiegers, *Peer Reviews in Software: A Practical Guide* [17*]. This book provides clear, succinct explanations of different peer review methods distinguished by level of formality and effectiveness. It provides pragmatic guidance for implementing the methods and for determining which methods are appropriate for given circumstances.

REFERENCES

- [1*] C. Y. Laporte, A. April, *Software Quality Assurance*, IEEE Press, 2018.
- [2] P. B. Crosby, *Quality Is Free*, McGraw-Hill, 1979.
- [3] W. Humphrey, *Managing the Software Process*, Addison-Wesley, 1989.
- [4] ISO/IEC, “ISO/IEC 25010:2011 Systems and Software Engineering — Systems and Software Quality Requirements

and Evaluation (SQuaRE) — Systems and Software Quality Models,” ed., 2011.

[5*] IEEE CS/ACM Joint Task Force on Software Engineering Ethics and Professional Practices, “Software Engineering Code of Ethics and Professional Practice
<https://www.computer.org/education/code-of-ethics>.

- [6] IEEE, “IEEE 730 Standard for Software Quality Assurance Processes,” ed., IEEE, 2014.
- [7*] I. Sommerville, *Software Engineering*, 10th ed., New York: Addison-Wesley, 2016.
- [8] RTCA, “DO-178C, Software Considerations in Airborne Systems and Equipment Certification,” ed., January 5, 2012. Also known as ED-12C in EUROCAE.
- [9] ISO/IEC, “ISO/IEC 15026-1:2019 Systems and Software Engineering — Systems and Software Assurance — Part 1: Concepts and Vocabulary,” ed., ISO/IEC, 2019.
- [10] ISO, “ISO 9000:2015 Quality Management Systems — Fundamentals and Vocabulary,” ed., ISO, 2015.
- [11] IEEE, “IEEE Std. 1012:2016, Standard for System and Software Verification and Validation,” IEEE, 2016.
- [12] ISO/IEC 20246:2017, “Software and systems engineering — Work product reviews,” ed., 2017.
- [13*] K. E. Wiegers, *Software Requirements*, 3rd ed., Redmond, WA: Microsoft Press, 2013.
- [14] IEEE/ISO/IEC, “IEEE/ISO/IEC 24765: Systems and Software Engineering — Vocabulary,” 1st ed., 2017.
- [15] N. Leveson, *Safeware: System Safety and Computers*, Addison-Wesley Professional, 1995.
- [16] T. Gilb and D. Graham, *Software Inspection*, Addison-Wesley Professional, 1993.
- [17*] K. Wiegers, *Peer Reviews in Software: A Practical Guide*, Addison-Wesley Professional, 2001.
- [18] BS EN 50128:2011+A2:2020, “Standard for Railway Applications – Communications, Signaling and Processing Systems – Software for Railway Control and Protection Systems,” British-Adopted European Standard, August 10, 2020.
- [19] K. Iberle, *They don't care about quality*, proceedings of STAR East, Orlando, United States, 2013, available at <https://kiberle.com/publications/>.
- [20] D. Wallace, L. M. Ippolito, B. B. Cuthill, *Reference Information for the Software Verification and Validation Process*, National Institute of Standards and Technology (NIST), U.D. Department of Commerce, Special Publication 500-234, 1996.
- [21] IEC 60300-1:2014, “Dependability Management — Part 1: Guidance for Management and Application,” version 3, September 25, 2014.
- [22] ISO/IEC TR 29110-1:2016, “System and Software Engineering — Lifecycle Profiles for Very Small Entities (VSEs) — Part 1: Overview Dependability Management — Part 1: Guidance for Management,” 2nd ed., 2016-06.
- [23] ISO/IEC TS 33061:2021, “Information technology — Process assessment — Process Assessment Model for Software Life Cycle Processes,” 2021-04.
- [24] IEEE Std 15288.2:2014, “IEEE Standard for Technical Reviews and Audits on Defense Programs.”
- [25] A guide to the Project management Body of Knowledge, 7th edition, PMI, 2021, 368p.

CHAPTER 13

SOFTWARE SECURITY

ACRONYMS

CC	Common Criteria
SD LC	Secure development life cycle

INTRODUCTION

Security has become a significant issue in software development because of increasing malicious activity targeting computer systems. In addition to the usual correctness and reliability concerns, software developers must pay attention to the security of the software they develop. Secure software development builds security by following a set of established and/or recommended rules and practices. Secure software maintenance complements secure software development by ensuring that no security problems are introduced during software maintenance.

BREAKDOWN OF TOPICS FOR SOFTWARE SECURITY

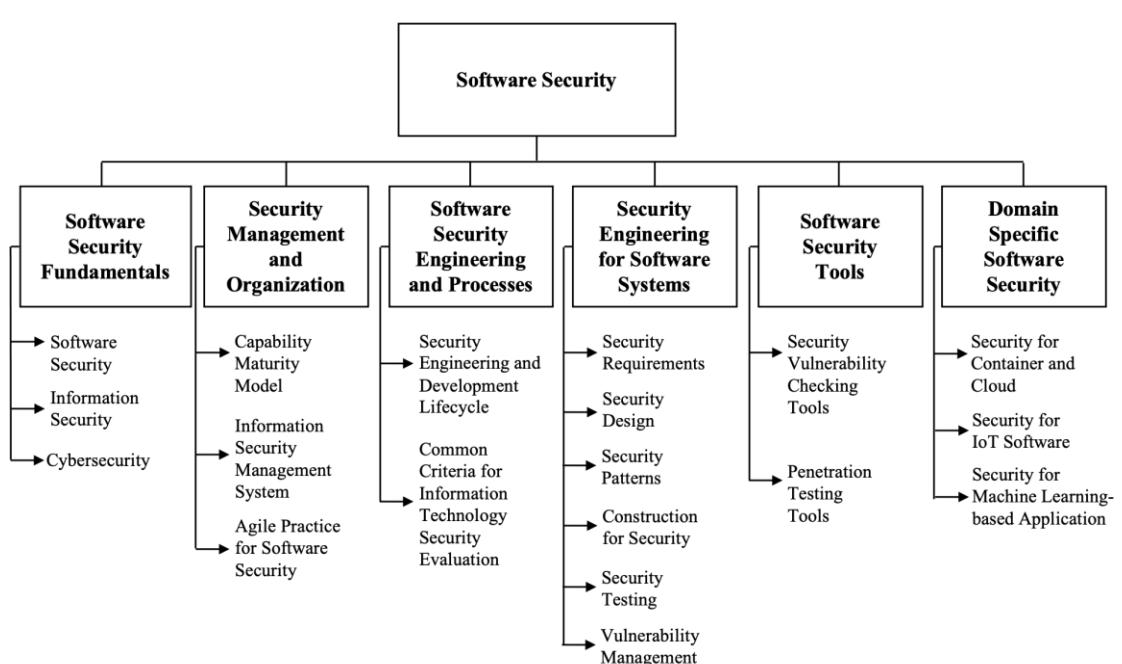


Figure 13.1. The Breakdown of Topics for the Software Security KA

1. Software Security Fundamentals [37, 9]

A generally accepted belief about software security is that it is much better to design security into software than to patch it in after the software is developed. To design security into software, one must consider every development life cycle stage. Secure software development involves software requirements security, software design security, software construction security and software testing security. In addition, security must be considered during software maintenance, as security faults and loopholes can be and often are introduced during maintenance.

1.1. Software Security [10]*

Security is a product quality characteristic representing the degree to which a product or system protects information and data so that persons or other products or systems have data access appropriate to their types and levels of authorization [10]. (For more information about product quality, refer to the Software Quality KA.)

1.2. Information Security [11]*

Information security preserves confidentiality, integrity and availability of information. Other properties, such as authenticity, accountability, non-repudiation and reliability can also be involved [11]. *Confidentiality* is the property of ensuring that information is not disclosed to unauthorized individuals, entities or processes. *Integrity* is the property of accuracy and completeness. *Availability* is the property of being accessible and usable on demand by an authorized entity. Software engineers should define the security properties of their software

and maintain them throughout the software development life cycle.

1.3. Cybersecurity [12]/[38]*

Cybersecurity addresses security issues in cyberspace, including the following:

- Social engineering attacks
- Hacking
- The proliferation of malicious software (*malware*)
- Spyware
- Other potentially unwanted software [12]

Software engineers should consider the mitigation of such risks as part of software development.

2. Security Management and Organization [1*, c7][13]

Security governance and management are most effective when they are systematic; in other words, when they are woven into the culture and fabric of organizational behaviors and actions. Project managers need to elevate software security from a stand-alone technical concern to an enterprise issue [1].

2.1. Capability Maturity Model [3, c22][14]*

Many organizations practice security engineering in the development of computer programs, including operating systems, functions that manage and enforce security, packaged software products, middleware, and applications. Therefore, a diverse array of individuals must know how to apply appropriate methods and practices, including product developers, service providers, system integrators, system administrators and even security specialists.

Systems Security Engineering — Capability Maturity Model (SSE-CMM), which helps measure the process capability of an organization that performs risk assessments [14], can be an important tool.

2.2. Information Security Management System [15*]

International Organization of Standardization/International Electrotechnical Commission (ISO/IEC) 27001:2013 specifies the requirements for establishing, implementing, maintaining and continually improving an information security management system (ISMS) within the organizational context [15]. Software that is part of an ISMS in an organization complies with the security policy. In other words, some software security properties come from the policy for ISMS.

2.3. Agile Practice for Software Security [4*,c15,c16]

Agile teams need to understand and adopt security practices and take more responsibility for their systems' security. Security professionals must learn to accept change, work faster and more iteratively, and think about security risks and how to manage risks in incremental terms. Finally, and most important, security needs to become an enabler instead of a blocker. The keys to a successful Agile security program are the involvement of the security team and developers, enablement, automation, and agility to keep up with Agile teams [4].

3. Software Security Engineering and Processes

3.1. Security Engineering and Secure Development Life Cycle (SDLC) [1*, c1][16*][36]

Software is only as secure as its development process. Security must be built into software engineering to ensure software security. The SDLC concept is one trend that aims to do this. SDLC uses a classical spiral model that views security holistically from the perspective of the software life cycle and ensures that security is inherent in software design and development, not an afterthought later in production. The SDLC process is claimed to reduce software maintenance costs and increase software reliability against security-related faults.

Recently, DevSecOps (meaning the integration of development, security and operations) has emerged. Beyond SDLC, DevSecOps includes an approach to culture, automation and platform design to make the software life cycle as Agile and responsible as Agile development and continuous integration (CI).

3.2. Common Criteria for Information Technology Security Evaluation [3*, c22, c25][34][35]

Security evaluation establishes confidence in the security functionality of IT products and the assurance measures applied to them. The evaluation results may help consumers determine whether IT products meet their security needs. ISO/IEC 15408, named Common Criteria (CC) for Information Technology Security Evaluation, is useful as a guide for developing, evaluating and/or procuring IT products with security functionality [34].

CC addresses the protection of assets from unauthorized disclosure, modification or loss of use. The categories of protection relating to these three types of security failure are commonly called *confidentiality*, *integrity* and *availability*, respectively.

4. Security Engineering for Software Systems

[1*,c1,c3][3*,c1,c3]

4.1. Security Requirements

[1*,c3][2*,c2][3*,c20,c30][18]

Software requirements security deals with clarifying and specifying security policy and objectives into software requirements, laying the foundation for security considerations in software development. Factors to consider in this phase include software requirements and threats or risks. The former refers to the specific functions required for security; the latter refers to the possible ways that software security is threatened.

4.2. Security Design

[1*,c4][2*,c5][3*,c20,c31][17,40]

Security design concerns how to prevent unauthorized disclosure, creation, change, deletion or denial of access to information and other resources. It also concerns how to tolerate security-related attacks or violations by limiting damage, continuing service, speeding repair and recovery, and failing and recovering securely. Access control is a fundamental concept of security; ensuring the proper use of cryptology is also important.

Software design security deals with the design of software modules that fit together to meet the security objectives specified in the security requirements. This step clarifies the details of security considerations and develops the specific steps for implementation. Factors considered may include frameworks and access modes that set up the overall security monitoring/enforcement strategies, as well as the individual policy enforcement mechanisms.

4.3. Security Patterns [1,c4][19, 20, 21]*

A *security pattern* describes a particular recurring security problem that arises in a specific context and presents a well-proven generic solution [21].

4.4. Construction for Security

[1*,c5][3*,c20,c31][22, 23, 24]

Software construction security concerns how to write programming code for specific situations to address security considerations. The term software construction security can mean different things to different people. It can mean the way a specific function is coded so that the code itself is secure, or it can mean the coding of security into software. Unfortunately, most people entangle the two meanings without distinction. One reason for such confusion is that it is unclear how to ensure a specific coding is secure. For example, in the C programming language, the expressions “`i<<1`” (shift the binary representation of `i`’s value to the left by one bit) and “`2*`” (multiply the value of variable `i` by constant 2) mean the same thing semantically, but do they have the same security ramifications?

The answer could be different for different combinations of ISAs and compilers. Because of this lack of understanding, software construction security — in its current state — mostly refers to the second aspect mentioned above: the coding of security into software. Coding of security into the software can be achieved by following recommended rules. A few such rules follow:

- Structure the process so that all sections requiring extra privileges are modules. The modules should be as small as possible and perform only the tasks that require those privileges.
- Ensure that any assumptions in the program are validated. If this is not

- possible, document them for the installers and maintainers so they know the assumptions attackers will try to invalidate.
- Ensure that the program does not share objects in memory with any other program.
- Check every function's error status. Do not recover unless neither the error's cause nor its effects affect any security considerations. The program should restore the state of the software to the state it had before the process began and then terminate.

Although there are no bulletproof ways to achieve secure software development, some general guidelines exist that can be helpful. These guidelines span every phase of the software development life cycle. The Computer Emergency Response Team (CERT) publishes reputable guidelines, and the following are its top 10 software security practices (the details can be found in [22]):

1. Validate input.
2. Heed compiler warnings.
3. Architect and design for security policies.
4. Keep it simple.
5. Default deny.
6. Adhere to the principle of least privilege.
7. Sanitize data sent to other software.
8. Practice defense in depth.
9. Use effective quality assurance techniques.
10. Adopt a software construction security standard.

4.5. Security Testing

[1*,c5]/[2*,c7]/[3*,c24,c31]/[26, 27]

Security testing ensures that the implemented software meets the security requirements. It also verifies that the software implementation contains none of the known vulnerabilities. Whereas general software testing methods can handle the former, the latter requires security-specific testing methods. (For more information about testing, please refer to the Software Testing KA.)

There are two approaches to security-specific testing. One is to detect programming language or implementation-specific vulnerabilities by static analysis of the source code, which can be automated using tools. The other is the *penetration test*, also known as the *ethical hacking test*. It detects vulnerabilities in software behavior. It can be automated using tools such as web application scanners and fuzzing tools. Security experts who are skilled in the latest security in the application domain should conduct these tests.

4.6. Vulnerability Management

[1*,c5]/[3*,c24]/[28,29, 30]

Using sound coding practices can help substantially reduce software defects commonly introduced during implementation [1]. Such common security defects are categorized and shared with databases: the Common Vulnerabilities and Exposures (CVE) [28], Common Weakness Enumeration (CWE) [29], and Common Attack Pattern Enumeration and Classification (CAPEC) [30]. Programmers can refer to these databases for security implementation, and some tools are available to check common vulnerabilities in codes.

5. Software Security Tools

5.1. Security Vulnerability Checking Tools

[1*,c6]/[25]

Some source code analysis tools check codes to detect security problems such as security vulnerabilities. However, identifying security vulnerabilities is complicated by the fact that they often appear in hard-to-produce software states or crop up in unusual circumstances [1]. Security analysis tools can help, but they cannot find all vulnerabilities.

5.2. Penetration Testing Tools [2,c4]*

Penetration testing tests a system in its final production environment. These tools submit malformed, malicious and random data to a system's entry points in an attempt reveal faults — a process commonly called *fuzzing* [2].

6. Domain-Specific Software Security

6.1. Security for Container and Cloud [31]

Cloud infrastructure and services are often inexpensive and easy to provision, which can quickly lead to having many assets strewn all over the world and forgotten. These forgotten assets are like a ticking time bomb, waiting to explode into a security incident [31].

One important difference with cloud environments is that physical assets and protection are generally not a concern. Developers can gleefully outsource asset tags, anti-tailgating, slab-to-slab barriers, placement of data center windows, cameras, and other

physical security and physical asset tracking controls [31].

6.2. Security for IoT Software [32,33]

As part of today's IoT (Internet of Things), systems are interconnected with many other devices, especially back-end systems suffering from all the well-known security flaws inherent in today's business IT. Attackers gaining access to business IT platforms, for instance, by exploiting browser vulnerabilities, will likely also gain access to weakly protected IoT industrial devices. This can cause severe damage, including safety incidents. Hence, the introduction of a massive number of end points from the consumer or industrial environment creates fertile ground for the exploitation of weak links. Hardening these end points, securing device-to-device communications, and ensuring device and information credibility in what until now have been closed, homogeneous systems present new challenges.

Comprehensive risk and threat analysis methods, as well as management tools for IoT platforms, are required [33].

6.3. Security for Machine Learning-Based Application [39,c8]

Although machine learning techniques are widely used in many systems, machine learning presents a specific vulnerability: Attackers can change the decisions of machine learning models. There are two kinds of attacks: model poisoning, which attacks training data, and evasion, which attacks inputs to trained models [39].

MATRIX OF TOPICS VS. REFERENCE MATERIAL

Topic	Allen et al. 2018 [1*]	McGrow 2006 [2*]	Bishop 2019 [3*]	Bell 2017 [4*]
1. Software Security Fundamentals				
1.1. Software Security				
1.2. Information Security				
1.3. Cybersecurity			Ch. 23	
2. Security Management and Organization	Ch. 7			
2.1. Capability Maturity Model			Ch. 22	
2.2. Information Security Management System				
2.3. Agile Practice for Software Security				Ch. 15, Ch. 16
3. Software Security Engineering and Processes				Ch. 9
3.1. Security Engineering and Secure Development Life Cycle	Ch. 1			Ch. 4
3.2. Common Criteria for Information Technology Security Evaluation			Ch. 22, Ch. 25	

4. Security Engineering for Software Systems			Ch. 1, Ch. 15, Ch. 1, Ch. 3	
4.1. Security Requirements	Ch. 3	Ch. 2	Ch. 20, Ch. 31	Ch. 5, Ch. 8
4.2. Security Design	Ch. 4	Ch. 5	Ch. 20, Ch. 31	Ch. 8
4.3. Security Patterns	Ch. 4			
4.4. Construction for Security	Ch. 5		Ch. 20, Ch. 31	
4.5. Security Testing	Ch. 5		Ch. 24, Ch. 31	Ch. 10, Ch. 11
4.6. Vulnerability Management			Ch. 24	Ch. 6
5. Software Security Tools				
5.1. Security Vulnerability Checking Tools	Ch. 6			Ch. 6
5.2. Penetration Testing Tools	Ch. 4		Ch. 31	Ch. 11, Ch. 12
6. Domain-Specific Software Security				
6.1. Security for Container and Cloud				

6.2. Security for IoT Software				
6.3. Security for Machine Learning-Based Application				

FURTHER READING

- [A] John Viega, *Building Secure Software: How to Avoid Security Problems the Right Way*, Addison-Wesley, 2011.
- [B] Loren Kohnfelder, *Designing Secure Software: A Guide for Developers*, No Starch Press, 2021.
- [C] C. Warren Axelrod, *Engineering Safe and Secure Software Systems*, Artech House Publishers, 2012.

REFERENCES

- [1*] Julia H. Allen, Sean J. Barnum, Robert J. Ellison, Gary McGraw, and Nancy R. Mead, *Software Security Engineering: A Guide for Project Managers*, Addison-Wesley Professional, 2008.
- [2*] Gary McGraw, *Software Security: Building Security In*, Addison-Wesley Professional, 2006.
- [3*] Matt Bishop, *Computer Security*, 2nd Edition, Addison-Wesley Professional, 2019.
- [4*] Laura Bell, Michael Brunton-Spall, Rich Smith, Jim Bird, *Agile Application Security*, O'Reilly, 2017.
- [5] Tony Hsiang-Chih Hsu, *Hands-On Security in DevOps: Ensure continuous security, deployment, and delivery with DevSecOps*, Packt Publishing, 2018.
- [6] Tony Hsiang-Chih Hsu, *Practical Security Automation and Testing: Tools and techniques for automated security scanning and testing in DevSecOps*, Packt Publishing, 2019.
- [7] Glenn Wilson, *DevSecOps: A leader's guide to producing secure software without compromising flow, feedback and continuous improvement*, Rethink Press, 2020.
- [8] Liz Rice, *Container Security: Fundamental Technology Concepts That Protect Containerized Applications*, O'Reilly & Associates Inc., 2020.
- [9] ISO/IEC/JTC1 SC27 Standards: Trustworthiness, Cryptography, Data security, Cryptography, Security evaluation and testing, Security control, Identity management and privacy technologies.
- [10*] ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models.

- [11*] ISO/IEC 27000:2018 Information technology — Security techniques — Information security management systems — Overview and vocabulary.
- [12*] ISO/IEC 27032:2012 Information technology — Security techniques — Guidelines for cybersecurity.
- [13] ISO/IEC 19770-1:2017 Information technology — IT asset management — Part 1: IT asset management systems — Requirements.
- [14] ISO/IEC 21827:2008 Information technology — Security techniques — Systems Security Engineering — Capability Maturity Model (SSE-CMM).
- [15*] ISO/IEC 27001:2013 Information technology — Security techniques — Information security management systems — Requirements.
- [16] Michael Howard and Steve Lipner, *The Security Development Lifecycle*, Microsoft Press, 2006.
- [17] Frank Swiderski, Window Snyder, *Threat Modeling: Design for Security*, Wiley, 2014.
- [18] Donald Firesmith, “Security use cases,” *Journal of Object Technology*, Vol. 2, No. 1, pp. 53-64, 2003.
- [19] Eduardo Fernandez-Buglioni, *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*, Wiley, 2013.
- [20] Christopher Nagappan, Ramesh Lai, Ray Steel, *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*, Prentice Hall, 2005.
- [21] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, Peter Sommerlad, *Security Patterns: Integrating Security and Systems Engineering*, Wiley, 2006.
- [22] Robert C. Seacord, *The CERT C Secure Coding Standard*, Addison-Wesley Professional, 2008.
- [23] Robert C. Seacord, *Secure Coding in C and C++*, Addison-Wesley Professional, 2013.
- [24] David Long, Fred Mohindra, Dhruv Seacord, Robert C. Sutherland, Dean F. Svoboda, *The CERT Oracle Secure Coding Standard for Java*, 2011.
- [25] Jon Erickson, *Hacking: The Art of Exploitation*, 2nd Edition, No Starch Press, 2008.
- [26] Karen Scarfone, Murugiah Souppaya, Amanda Cody, Angela Orebaugh, *Technical Guide to Information Security Testing and Assessment*, NIST SP800-115, 2008.
- [27] PCI Security Standards Council, PCI DSS: Payment Card Industry Data Security Standard, Version 3.2, 2017.
- [28] MITRE, “Common Vulnerabilities and Exposures (CVE),” <https://cve.mitre.org/>.
- [29] MITRE, “Common Weakness Enumeration (CWE),” <https://cwe.mitre.org/>.

- [30] MITRE, “Common Attack Pattern Enumeration and Classification (CAPEC),” <https://capec.mitre.org/>.
- [31] Chris Dotson, *Practical Cloud Security*, O’Reilly, 2019.
- [32] “Internet of Things Security Best Practices,” IEEE, 2017, <https://internetinitiative.ieee.org/resources/reports-presentations-publications>.
- [33] “IoT 2020: Smart and secure IoT platform,” IEC, 2016, <https://www.iec.ch/basecamp/iot-2020-smart-and-secure-iot-platform>.
- [34] ISO/IEC 15408-1:2009 Information technology — Security techniques — Evaluation criteria for IT security — Part 1: Introduction and general model.
- [35] ISO/IEC 18045:2008 Information technology — Security techniques — Methodology for IT security evaluation.
- [36] DoD Enterprise DevSecOps, <https://software.af.mil/dsop/documents/>.
- [37] Chuck Easttom, *Computer Security Fundamentals*, 4th Edition, Pearson IT Certification, 2019.
- [38] Yuri Diogenes, Erdal Ozkaya, *Cybersecurity — Attack and Defense Strategies*, Second Edition, Packt Publishing, 2019.
- [39] Clarence Chio, David Freeman, *Machine Learning and Security: Protecting Systems with Data and Algorithms*, O’Reilly, 2018.
- [40] Ian Sommerville, *Software Engineering*, 10th ed., Addison-Wesley, 2016.

CHAPTER 14

SOFTWARE ENGINEERING PROFESSIONAL PRACTICE

ACRONYMS

ACM	Association for Computing Machinery
CCPA	The California Consumer Privacy Act
EEA	European Economic Area
ENAE	European Network for Accreditation of Engineering Education
EU	European Union
GDPR	The General Data Protection Regulation
IEA	International Engineering Alliance
IEC	International Electrotechnical Commission
IEEE-CS	Institute for Electrical and Electronics Engineers — Computer Society
IFIP	International Federation for Information Processing
IP	Intellectual property
ISO	International Organization for Standardization
NDA	Nondisclosure agreement
UI/UX	User interface/user experience
WIPO	World Intellectual Property Organization
WTO	World Trade Organization

responsible and ethical manner. Because of the widespread applications of software products in social and personal life, software product quality can profoundly affect personal well-being and societal harmony. Software engineers must handle unique engineering problems to produce software with known characteristics and reliability. This requirement calls for software engineers who possess the proper knowledge, skills, training, and experience in professional practice.

Professional practice refers to a way of conducting services to achieve certain standards or criteria in both the process of performing a service and the end product resulting from the service. These standards and criteria can include both technical and non-technical aspects. The concept of professional practice is especially applicable to professions with a generally accepted body of knowledge; code of ethics and professional conduct with penalties for violations; accepted processes for accreditation, certification, qualification, and licensing; and professional societies to provide and administer all these. Admission to these professional societies is often predicated on a prescribed combination of education and experience.

A software engineer maintains professional practice by performing all work following generally accepted practices, standards, and guidelines set forth by the applicable professional society, such as the Association for Computing Machinery (ACM), Institute for Electrical and Electronics Engineers (IEEE), or International Federation for Information Processing (IFIP). The Institute for Electrical and Electronics Engineers — Computer Society (IEEE-CS), International Organization for Standardization/International Electrotechnical Commission (ISO/IEC), and ISO/IEC/IEEE provide internationally accepted software engineering standards. All of these standards and guidelines comprise the foundation of the professional practice of software engineering.

INTRODUCTION

The Software Engineering Professional Practice knowledge area (KA) is concerned with the knowledge, skills, and attitudes software engineers must possess to practice software engineering in a professional,

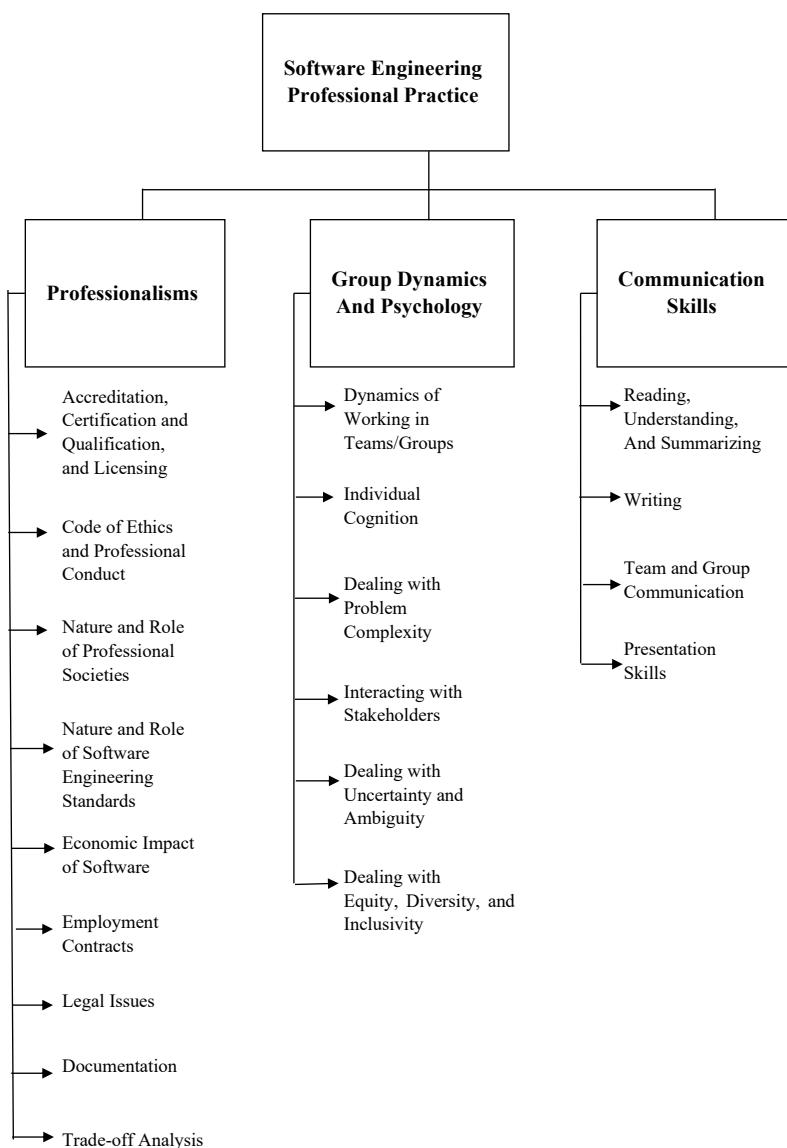


Figure 14.1. Breakdown of Topics for the Software Engineering Professional Practice KA

BREAKDOWN OF TOPICS FOR SOFTWARE ENGINEERING PROFESSIONAL PRACTICE

The Software Engineering Professional Practice KA's breakdown of topics is shown in Figure 14.1.

The subareas presented in this KA are professionalism, group dynamics and psychology, and communication skills.

1. Professionalism

A software engineer displays professionalism notably by adhering to a code of ethics and professional conduct and to standards and practices established by the engineer's professional community.

One or more professional societies often represent a professional community, and this is the case for the engineering community. Those societies publish codes of ethics and professional conduct as well as criteria for admittance to the community. Those criteria form the basis for accreditation and licensing activities and may determine engineering competence or negligence.

As software is used more widely and deeply in society, stakeholders' requirements have likewise become wider and deeper. And as software has become socially vital, software engineers have worked to base the user interface/user experience (UI/UX) on socially inclusive concepts.

1.1. Accreditation, Certification and Qualification, and Licensing [1, cls4-s5, cls10] [2] [4*, c12s10] [6] [7] [8] [9]*

1.1.1. Accreditation

Accreditation certifies an organization's competency, authority, or credibility.

Accredited schools or programs have shown that they adhere to particular standards and maintain certain qualities. In many countries, the basic means by which engineers acquire knowledge is by completing an accredited course of study. Often, the accreditation process is independent of the government and is performed by private membership associations. There are two major global accreditation organizations. One is the International Engineering Alliance (IEA), of which the Washington Accord is a constituent. The other is the European Network for Accreditation of Engineering Education (ENAE), which administers EUR-ACE®, the label awarded to engineering degree programs at the bachelor's and master's levels, listed by the European Commission.

Although the accreditation process might differ for each country and jurisdiction, the general meaning is the same. Accreditation of an institution's course of study means "the accreditation body recognizes an educational institution as maintaining standards that qualify the graduates for admission to higher or more specialized institutions or professional practice."

1.1.2. Certification and Qualification

ISO/IEC 24773-1 Software and Systems Engineering — Certification of Software and Systems Engineering Professionals — Part 1: General Requirements define certification and qualification. Certification contains recertification. Qualification is similar to certification but does not require requalification. Certification refers to the confirmation of a person's particular characteristics. A common type of certification is professional certification, which certifies a person as being able to complete an activity in a certain discipline at a stated level of competency. Professional certification can verify the holder's ability to meet professional standards and to apply professional judgment in solving or addressing problems. Professional certification can also verify prescribed knowledge, mastery of best practices and proven methodologies, and professional experience.

An engineer usually obtains certification by passing an examination in addition to meeting

other experience-based criteria. Nongovernmental organizations, such as professional societies, often administer these examinations. In software engineering, certification testifies to one's capability as a software engineer.

The IEEE-CS has enacted certification programs [9]. Currently, they are classified in the following levels, Associate Software Developer, Professional Software Developer, and Professional Software Engineering Master. The qualification and certification programs are designed to confirm a software engineer's knowledge of standard software engineering practices and to advance the engineer's career. A lack of qualification or certification does not exclude the individual from working as a software engineer. Qualification or certification in software engineering is voluntary. Most software engineers are not qualified or certified under any program

1.1.3. Licensing

Licensing authorizes a person to perform certain activities and take responsibility for resultant engineering products. The noun *license* refers to both that authorization and the document recording that authorization. Governmental authorities or statutory bodies usually issue licenses.

Obtaining a license to practice requires an individual to meet a certain standard at a certain ability to practice or operate. Sometimes an entry-level requirement sets the minimum skills and capabilities to practice, and as the professional moves through their career, the required skills and capabilities change and evolve.

Engineers are licensed to protect the public from unqualified individuals. In some countries, no one can practice as a professional engineer unless licensed; further, no company may offer "engineering services" unless at least one licensed engineer is employed there.

1.2. Codes of Ethics and Professional Conduct

[1*, cls7-cls9, c10s2, Appendix] [3*, c8] [4*, cls2] [5*, c33] [10] [11] [13*]

Codes of ethics and professional conduct describe the values and behavior that an engineer's professional practice and decisions should embody. The professional community establishes a code of ethics and professional conduct. This code exists in the context of societal norms and local laws and is adjusted to agree with those norms and laws as needed. A code of ethics and professional conduct can offer guidance in the face of conflicting imperatives. More than one such code serves the professional engineering community. For example, in 1999, IEEE-CS and ACM launched a joint Software Engineering Ethics and Professional Practices Task Force to publish a code of ethics. In 2018, ACM published its ACM Code of Ethics and Professional Conduct, and in 2020, IEEE published a revision of its Code of Ethics which was originally approved in 1912. Then, in 2021, IFIP published its Code of Ethics and Professional Conduct, adapted from ACM's Code of Ethics and Professional Conduct.

Once established, codes of ethics and professional conduct are enforced by the profession, as represented by professional societies or by a statutory body. Violations may be acts of commission, such as concealing inadequate work, disclosing confidential information, falsifying information, or misrepresenting abilities. They may also occur through omission, including failure to disclose risks or provide important information, failure to give proper credit or acknowledge references, and failure to represent client interests. Violations of a code of ethics and professional conduct may result in penalties and possible expulsion from professional status.

Software engineers shall commit themselves to making the analysis, specification, design, development, testing, and maintenance of software a beneficial and respected profession. Following their commitment to the health, safety, and welfare of the public, software engineers shall adhere to the eight principles concerning the public, client and employer,

product, judgment, management, profession, colleagues, and self.

Since the code of ethics and professional conduct may be introduced, modified, or replaced at any time, individual software engineers are responsible for continuing their studies to stay current in their professional practice.

1.3. Nature and Role of Professional Societies [1*, c2s3] [4*, c1s2] [5*, c35s1]

Professional societies comprise a mix of practitioners and academics. These societies define, advance, and regulate their corresponding professions. Professional societies help establish professional standards as well as codes of ethics and professional conduct. They also engage in related activities, which include the following:

- Establishing and promulgating a body of generally accepted knowledge
- Providing the basis for licensing, certifying, and accrediting
- Dispensing disciplinary actions
- Advancing the profession through conferences, training, publications, and standards

Participation in professional societies assists individual engineers in maintaining and sharpening their professional knowledge and relevancy and in expanding and maintaining their professional network.

1.4. Nature and Role of Software Engineering Standards [1*, c10s2] [2] [4*] [5*, c32s6]

Software engineering standards cover a remarkable variety of topics. They provide guidelines for the practice of software engineering and for processes to be used during the development, maintenance, and support of software. By establishing a common body of knowledge and experience, software engineering standards establish a basis on which further guidelines may be developed. Appendix B of this *Guide* presents IEEE, ISO/IEC, and ISO/IEC/IEEE software engineering standards that support this *Guide's* KAs.

Standards are valuable sources of information about requirements and other guidance that can support software engineers in everyday activities. Adherence to standards promotes discipline by enumerating minimal characteristics of products and practices. That discipline helps mitigate subconscious assumptions or overconfidence in a design. For these reasons, organizations performing software engineering activities often include conformance to standards as part of their organizational policies.

1.5. Economic Impact of Software [3*, c1s1, c10s8] [4*, c1s1] [13*]

The software has economic effects at the individual, business, and societal levels. For example, software “success” may be determined by a product’s suitability for a recognized problem and by its effectiveness when applied to that problem. At the individual level, an engineer’s continuing employment may depend on their ability and willingness to interpret and execute tasks in meeting customers’ or employers’ needs and expectations. The customer’s or the employer’s financial situation may be positively or negatively affected by software purchases.

At the business level, software properly applied to a problem can eliminate months of work and translate to elevated profits or more effective organizations. Organizations that acquire or provide successful software may become a boon to the society in which they operate by providing both employment and improved services. However, the software’s development or acquisition costs can be considerable, like those of any major acquisition.

At the societal level, direct impacts of software success or failure include the avoidance or experience of accidents, interruptions, and loss of service. Indirect impacts include the success or failure of the organization that acquired or produced the software, increased or decreased societal productivity, harmonious or disruptive social order, and even the saving or loss of property or life. In addition, as digitalization progresses, easier and faster access to the information needed may bring higher social value.

1.6. Employment Contracts [1, c6, c7] [10] [11] [12]*

Software engineering services may be provided under a variety of client-engineer relationships. For example, the work may be done through a company-to-customer supplier arrangement, an engineer-to-customer consultancy arrangement, a direct-hire, or even through volunteering. In these situations, the customer and supplier agree that a product or service will be provided in return for some consideration. Here, we are most concerned with engineer-to-customer arrangements and their attendant agreements or contracts, whether they are of the direct-hire or consultant variety, and the issues they typically address.

A common concern in software engineering contracts is confidentiality. Employers derive commercial advantage from intellectual property (IP), so they strive to protect that property from disclosure. Therefore, software engineers are often required to sign nondisclosure agreements (NDA) or IP agreements as a precondition to working. These agreements typically apply to information the software engineer could gain only through association with the customer. The terms of these agreements may extend past the association's termination.

Another concern is IP ownership. Rights to software engineering assets — products, innovations, inventions, discoveries, and ideas — may reside with the employer or customer, under explicit contract terms or relevant laws, if those assets are obtained during the software engineer's relationship with that employer or customer. Contracts differ in the ownership of assets created using non-employer-owned equipment or information.

Finally, contracts can also specify, among other elements:

- The location at which work is performed
- Standards to which that work will be held
- The system configuration used for development

- Limitations of the software engineer's and employer's liability
- A communication matrix and/or escalation plan
- Administrative details such as rates, frequency of compensation, working hours, and working conditions

1.7. Legal Issues [1, c6, c11] [2] [3*, c5s3–c5s4] [4*, c12s3, c13s2]*

Legal issues surrounding software engineering professional practice include matters related to standards, trademarks, patents, copyrights, trade secrets, professional liability, legal requirements, trade compliance, cybercrime, and data privacy. It is therefore beneficial to know these issues and their applicability. In addition, legal issues are jurisdictionally based, so software engineers must consult attorneys who specialize in the type and jurisdiction of any identified legal issues.

1.7.1. Standards

Adherence to standards provides a defense from legal action or allegations of malpractice.

1.7.2. Trademarks

A trademark relates to any word, name, symbol, or device used in business transactions. It is used “to indicate the source or origin of the goods.” Trademark protection protects names, logos, images, and packaging. However, if a name, image, or other trademarked asset becomes a generic term, trademark protection is nullified.

The World Intellectual Property Organization (WIPO) is the authority that frames the rules and regulations on trademarks. WIPO is the United Nations agency dedicated to protecting the use of IP as a means of stimulating innovation and creativity.

1.7.3. Patents

Patents protect an inventor's right to manufacture and sell an idea. A patent consists of exclusive rights granted by a sovereign government to an individual, group of individuals, or organization for a limited period.

Patents are an old form of idea-ownership protection and date to the 15th century.

Application for a patent entail keeping and producing careful records of the process that led to the invention. In addition, patent attorneys help write patent disclosure claims in a manner most likely to protect the software engineer's rights. Note that if inventions are made during a software engineering contract, ownership may belong to the employer or customer or be jointly held rather than belong to the software engineer.

Rules vary concerning what is and what is not patentable. In many countries, software code is not patentable, but software algorithms may be. Existing and filed patent applications can be found at WIPO.

1.7.4. Copyrights

Most governments give exclusive rights of an original work to its creator, usually for a limited time, enacted as copyright. Copyrights protect the way an idea is presented—not the idea itself. For example, they may protect the particular wording of an account of a historical event, whereas the event itself is not protected. Copyrights are long-term and renewable. As a form of IP, they date to the 17th century.

1.7.5. Trade Secrets

In many countries, an intellectual asset such as a formula, algorithm, process, design, method, pattern, instrument, or compilation of information may be considered a *trade secret*, provided the asset is not generally known and may provide a business with some economic advantage. The “trade secret” designation provides legal protection if the asset is stolen. This protection is not subject to a time limit. However, if another party derives or discovers the same asset legally, then the asset is no longer protected and the other party will also possess all rights to use it.

1.7.6. Professional Liability

It is common for software engineers to be concerned with professional liability matters. As engineers provide services to a client or employer, it is crucial that they adhere to

standards and generally accepted practices to protect themselves against allegations of or proceedings related to malpractice, negligence, or incompetence.

For engineers (including software engineers), professional liability is related to product liability. Under the laws and rules of their jurisdiction, engineers may be held accountable for failing to fully and conscientiously follow recommended practice; this is known as *negligence*. They may also be subject to laws governing *strict liability* and implied or express warranty, where, by selling the product, the engineer is held to warrant that the product is both suitable and safe for use. In some countries (e.g., in the US), *privity* (a doctrine under which one can sue only the person selling the product) is no longer a defense against liability actions.

Legal suits for liability can be brought under tort law in the US, allowing anyone who is harmed to recover their loss even if no guarantees were made. Because it is difficult to measure the suitability or safety of software, failure to take due care can be used to prove negligence on the part of software engineers. Engineers can defend themselves against such an allegation by showing that they followed standards and generally accepted practices in developing the product.

1.7.7. Legal Requirements

Software engineers must operate within local, national and international legal frameworks. Therefore, software engineers must know the legal requirements for the following:

- Registration and licensing, including examination, education, experience, and training requirements
- Contractual agreements
- Noncontractual legalities, such as those governing liability

Basic information on the international legal framework can be accessed from the World Trade Organization (WTO).

1.7.8. Trade Compliance

All software professionals must be aware of legal restrictions on the import, export, or re-export of goods, services, and technology in the jurisdictions in which they work. Such rules often concern export controls and classification; transfer of goods; acquisition of necessary governmental licenses for foreign use of hardware and software; services and technology by sanctioned nations, enterprises, or individual entities; and import restrictions and duties. Trade experts should be consulted for detailed compliance guidance.

1.7.9. Cybercrime

Cybercrime refers to any crime that involves a computer, computer software, computer networks, or embedded software controlling a system. The computer or software may have been used in the commission of a crime or have been the target of the crime. This category of crime includes fraud, unauthorized access, spam, obscene or offensive content, threats, harassment, theft of sensitive personal data or trade secrets, and use of one computer to damage or infiltrate other computers and automated system controls.

Computer and software users commit fraud by altering electronic data to facilitate illegal activity. Forms of unauthorized access include hacking, eavesdropping, and using computer systems in a way that is concealed from their owners. Many countries have laws that specifically cover cybercrimes, but many do not have effective statutes, making cybercrime difficult to prosecute in some cases. The software engineer has a professional obligation to consider the threat of cybercrime and to consider how the software system's security will protect the software and user information from accidental or malicious access, use, modification, destruction, or disclosure.

Dark patterns are deceptive UI/UX interactions designed to mislead or trick users into making them do something they may not want to do. These patterns do not have the users' interests in mind and aim for exploitability rather than usability. Creating dark patterns is not good ethical practice. Software engineers should be responsible for

their actions and be transparent with users instead of manipulating them.

1.7.10. Data Privacy

Software engineers should know that data privacy is a key legal requirement in many countries. The General Data Protection Regulation (GDPR), adopted on April 14, 2016, and enforceable since May 25, 2018, regulates data protection and privacy in the European Union (EU) and the European Economic Area (EEA). It also addresses the transfer of personal data outside the EU and EEA areas. The GDPR's primary aim is to enhance individuals' control and rights over their data and to simplify the regulatory environment for international business.

The regulation became a model for many national laws outside the EU, including the UK, Chile, Japan, Brazil, South Korea, Argentina, and Kenya. The California Consumer Privacy Act (CCPA), adopted on June 28, 2018, has many similarities with the GDPR.

1.8. Documentation [1, c10s5.8] [3*, c1s5] [4*] [5*, c32]*

Providing clear, thorough, and accurate documentation is the responsibility of each software engineer. The adequacy of documentation is judged according to different criteria, based on stakeholder needs. Good documentation complies with accepted standards and guidelines. In particular, software engineers should document the following:

- Relevant facts
- Significant risks and trade-offs
- Warnings of undesirable or dangerous consequences from the use or misuse of the software
- Relevant information pertaining to attribute, license type, and sourcing

Software engineers should avoid:

- Certifying or approving unacceptable products
- Disclosing confidential information
- Falsifying facts or data

In addition, software engineers and their managers should provide the following documentation for other elements of the software development organization to use:

- Software requirements specifications, software design documents, details on the software engineering tools used, software test specifications and results, and details about the adopted software engineering methods
- Problems encountered during the development process

For external stakeholders (customers, users, others), software documentation should provide the following:

- Information needed to determine whether the software is likely to meet customer and user needs
- Description of safe and unsafe use of the software
- Explanation of how to protect sensitive information created by or stored using the software
- Clear identification of warnings and critical procedures

Software use may include installation, operation, administration, and performance of other functions by various groups of users and support personnel. If the customer will acquire ownership of the software source code or the right to modify the code, the software engineer should provide documentation of the functional specifications, the software design, the test suite, and the necessary operating environment for the software. Documents should be kept for at least as long as the software product's life cycle or the time required by relevant organizational or regulatory requirements.

1.9. Trade-Off Analysis [3, c1s2, c10] [4*, c7s2, c13s4] [13*, c9s5.10]*

A software engineer often has to choose between alternative problem solutions. The outcome of these choices is determined by the software engineer's professional evaluation of each alternative's risks, costs, and benefits in cooperation with stakeholders. The software engineer's evaluation is called *trade-off analysis*.

Trade-off analysis notably identifies competing and complementary software requirements to prioritize the final requirements defining the software to be constructed. (See Requirements Negotiation in the Software Requirements KA and Determination and Negotiation of Requirements in the Software Engineering Management KA.)

When an ongoing software development project is late or over budget, a trade-off analysis is often conducted to decide which software requirements can be relaxed or dropped given the effects thereof. The first step in a trade-off analysis is establishing design goals (see Engineering Design in the Engineering Foundations KA) and setting the relative importance of those goals. This permits the identification of the solution that most nearly meets those goals; this means that the way the goals are stated is critically important.

Design goals may include minimizing monetary cost and maximizing reliability, performance, or other criteria on various dimensions. However, it is difficult to formulate a trade-off analysis of cost against risk, especially where primary production and secondary risk-based costs must be weighed against each other.

A software engineer must ethically conduct a trade-off analysis — notably by being objective and impartial when selecting criteria for comparing alternative problem solutions and assigning weights or importance to these criteria. In addition, any conflict of interest must be disclosed upfront.

2. Group Dynamics and Psychology

Engineering work is often conducted in teams. A software engineer should interact cooperatively and constructively with others to first determine and then meet needs and expectations. Knowledge of group dynamics and psychology is an asset when interacting with customers, coworkers, suppliers, and subordinates to solve software engineering problems.

2.1.

Dynamics of Working in Teams/Groups [3, c1s6] [14*, c1s3.5, c10]*

Software engineers must work with others. On the one hand, they work internally in engineering teams; on the other hand, they work with customers, members of the public, regulators, and other stakeholders. Performing teams — those who demonstrate a consistent quality of work and progress toward goals — are cohesive and possess a cooperative, honest and focused atmosphere. Individual and team goals are aligned so the members naturally commit to and feel ownership of shared outcomes.

Team members facilitate this atmosphere by being intellectually honest, using group thinking, admitting ignorance, and acknowledging mistakes. They share responsibility, rewards, and workload fairly. They communicate clearly and directly to one another and in documents and source code so information is accessible to everyone. Peer reviews about work products are framed in a constructive and nonpersonal way. (See Reviews and Audits in the Software Quality KA.) This allows all the members to pursue a continuous improvement and growth cycle without personal risk. Members of cohesive teams demonstrate respect for one another and their leader.

One point to emphasize is that software engineers must be able to work in multidisciplinary environments and varied application domains. Because software is everywhere, from phones to cars, it affects people's lives far beyond the more traditional concept of software made for information management in a business environment.

2.2. Individual Cognition [3*, c1s6.5] [5*, c33]

Engineers want to solve problems. Every engineer strives to solve problems effectively and efficiently. However, the limits and processes of individual cognition affect problem-solving. Individual cognition plays a prominent role in problem-solving in software engineering, in part because of the highly abstract nature of software itself.

An individual's (in particular, a software engineer's) ability to decompose a problem and

creatively develop a solution can be inhibited by the following:

- The need for more knowledge
- Subconscious assumptions
- The volume of data
- Fear of failure or the consequence of failure
- Culture, either of the application domain or the organization
- Lack of ability to express the problem
- Perceived working atmosphere
- The individual's emotional status

The effects of these inhibiting factors can be reduced by cultivating good problem-solving habits that minimize the impact of misleading assumptions. The ability to focus is crucial, as is intellectual humility. Both allow a software engineer to suspend personal considerations and consult with others freely, which is especially important when working in teams.

Engineers use basic methods to facilitate problem-solving. (See Problem-Solving Techniques in the Computing Foundations KA.) Breaking down problems and solving them one piece at a time reduces cognitive overload. By taking advantage of professional curiosity and pursuing continuous professional development, engineers gain skills and knowledge. Reading, networking, and experimenting with new tools, techniques and methods are all valid means of professional development.

2.3. Dealing with Problem Complexity [3*, c3s2] [4*, c1s1, c20s1-s5] [5*, c33]

Many, if not most, software engineering problems are too complex and difficult to address as a whole or to be tackled by individual software engineers. When such circumstances arise, engineers typically use teamwork and problem decomposition. (See Problem-Solving Techniques in the Computing Foundations KA.)

Teams work together to deal with large, complex problems by sharing burdens and drawing on one another's knowledge and creativity. When software engineers work in teams, individual engineers' different views and abilities complement one another and help build

a solution otherwise difficult to come by. Some teamwork examples in software engineering are pair programming (see Agile Methods in the Software Engineering Models and Methods KA) and code review (see Reviews and Audits in the Software Quality KA).

2.4. Interacting with Stakeholders [4]*

The success of a software engineering endeavor depends on positive interactions with stakeholders. Stakeholders should provide support, information, and feedback at all stages of the software life cycle. For example, during the early stages, it is critical to identify all stakeholders and discover how the product will affect them to properly capture a sufficient definition of stakeholder requirements.

In Agile software development, the involvement of stakeholders is even more essential than in other types of development. First, during development, stakeholders may provide feedback on specifications or early versions of the software, changes of priority, and clarification of detailed or new software requirements. Last, during software maintenance and until the end of product life, stakeholders can provide feedback on evolving or new requirements and problem reports so the software can be extended and improved. Clearly, regular stakeholder involvement will lead to a better application. It is vital to maintain open and productive communication with stakeholders during the software product's life cycle.

2.5.

Dealing with Uncertainty and Ambiguity [4, c4s1, c4s4, c11s5, c24s5] [14*, c9s4]*

As with engineers in other fields, software engineers must often deal with and resolve uncertainty and ambiguities while providing services and developing products. The software engineer must reduce or eliminate any lack of clarity that is an obstacle to performing work.

Often, uncertainty reflects a lack of knowledge. If that is the case, investigating the issue by reading formal sources such as textbooks and professional journals, interviewing stakeholders,

or consulting with teammates and peers can likely solve the problem.

When uncertainty or ambiguity cannot be overcome easily, software engineers or organizations may regard it as a project risk. When this is the case, work estimates or pricing are adjusted to mitigate the anticipated cost of addressing it. (See Risk Management in the Software Engineering Management KA.)

2.6. Dealing with Equity, Diversity, and Inclusivity [4] [14*, c10s7]*

The equity, diversity, and inclusivity environment can affect a group's dynamics. This is especially true when the group is geographically separated or communication is infrequent because such separation elevates the importance of each contact. Intercultural communication is even more difficult if the difference in time zones makes oral communication less frequent.

Multicultural environments are prevalent in software engineering, perhaps more than in other engineering fields, because of the strong trend of international outsourcing and the easy shipment of software components instantaneously around the globe. For example, it is common for a software project to be divided into pieces across national and cultural borders. It is also common for a software project team to consist of people from diverse cultural backgrounds.

For a software project to succeed, team members must embrace tolerance of different cultural and social norms, acknowledging that not all societies have the same social expectations. The support of leadership and management can facilitate tolerance and understanding. More frequent communication, including face-to-face meetings, can help mitigate geographical and cultural divisions, promote cohesiveness, and raise productivity. Also, communicating with teammates in their native language could be beneficial.

In the software industry, gender bias is still prevalent. Implementing broader recruiting strategies, specific and measurable performance

evaluation criteria, and transparent procedures for assigning compensation can reduce gender inequality in the software industry. These trends can contribute to building a diverse environment for all software engineers, regardless of their gender.

.

3. Communication Skills

A software engineer must communicate well, both orally and in reading and writing. To meet software requirements and deadlines, engineers must establish clear communication with customers, supervisors, coworkers, and suppliers. Optimal problem-solving is made possible through the ability to investigate, comprehend and summarize information. Customer product acceptance and safe product use depend on relevant training and documentation. The software engineer's career success is affected by consistently providing oral and written communication effectively and on time.

3.1. Reading, Understanding, and Summarizing [4, c4s5] [5*, c33s3]*

Software engineers must be able to read and understand technical material. Technical material includes reference books, manuals, research papers, and program source code.

Reading is not only a primary way of improving skills but also a way of gathering information for completing engineering goals. A software engineer sifts through accumulated information, focusing on the pieces that will be most helpful. Customers may request that a software engineer summarize the results of such information-gathering for them, simplifying or explaining it so that they can make the final choice among competing solutions.

Reading and comprehending source code are also components of information-gathering and problem-solving. For example, when engineers modify, extend or rewrite software, they must understand both its implementation, directly derived from the presented code, and its design, which must often be inferred.

3.2. Writing [3, c1s5] [4*, c4s2-s3]*

Software engineers can produce written products requested by customers or required by generally accepted practice. These written products may include source code, software project plans, software requirement documents, risk analyses, software design documents, software test plans, user manuals, technical reports and evaluations, justifications, diagrams and charts, and so forth.

Clear, concise writing is important because writing is often the primary method of communication among relevant parties. In all cases, written software engineering products must be accessible, understandable, and relevant to their intended audience.

3.3. Team and Group Communication [3, c1s6.8] [4*, c22s3] [5*, c27s1] [14*, c10s4]*

Effective communication among team and group members is essential to a collaborative software engineering effort. Stakeholders must be consulted; decisions must be made, and plans must be generated. The greater the number of team and group members, the greater the need to communicate.

However, the number of communication paths grows quadratically with the addition of each team member. Furthermore, team members are unlikely to communicate with anyone perceived to be removed from them by more than two degrees (levels). This problem can be more serious when software engineering endeavors or organizations are spread across national and continental borders.

Some communication can be accomplished in writing. Software documentation is a common substitute for direct interaction. Email is another, but although it is useful, it is not always enough. Also, if one receives too many messages, it becomes difficult to identify the important information. Increasingly, organizations are using enterprise collaboration tools to share information. In addition, electronic information stores, accessible to all team members for organizational policies, standards, common

engineering procedures, and project-specific information, can be beneficial.

Some software engineering teams focus on face-to-face interaction and promote such interaction through office space arrangements. Although private offices improve individual productivity, other arrangements, such as co-locating team members in physical or virtual spaces and providing communal work areas, can boost collaborative efforts.

3.4. Presentation Skills [3, c1s5] [4*, c22]
[14*, c10s7–c10s8]*

Software engineers rely on their presentation skills during software life cycle processes. For example, software engineers may walk customers and teammates through software requirements during the phase and conduct formal requirements reviews. (See Requirement Reviews in the Software Requirements KA.) During and after software design, software construction, and software maintenance, software engineers lead reviews, product walkthroughs (see Review and Audits in the Software Quality KA), and training. These require the ability to present technical information to groups and solicit ideas or feedback.

Therefore, the software engineer's ability to convey concepts effectively in a presentation influences product acceptance, management, and customer support. It also influences the ability of stakeholders to comprehend and assist in the product effort. This knowledge needs to be archived in slides, knowledge write-ups, technical white papers, and other material used for knowledge creation.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	Bott et al. 2000 [1*]	Voland, 2003 [3*]	Sommerville, 2016 [4*]	McConnel, 2004 [5*]	Tockey, 2004 [13*]	Fairley, 2009 [14*]
1. Professionalism						
1.1. Accreditation, Certification and Qualification, and Licensing	c1s5, c1s10		c12s10			
1.2. Codes of Ethics and Professional Conduct	c1s7-s9, c10s2, App		c1s2			
1.3. Nature and Role of Professional Societies	c2s3		c1s2	c35s1		
1.4. Nature and Role of Software Engineering Standards	c10s2,		*	c32s6		
1.5. Economic Impact of Software		c1s1, c10s8	c1s1		*	
1.6. Employment Contracts	c6, c7		*			
1.7. Legal Issues	c6, c11	c5s3-s4,	c12s3, c13s2			
1.8. Documentation	c10s5	c1s5	*	c32		
1.9. Trade-Off Analysis		c1s2, c10	c7s2, c13s4		c9s5-10	
2. Group Dynamics and Psychology						
2.1. Dynamics of Working in Teams/Groups		c1s6				c1s3-5, c10
2.2. Individual Cognition		c1s6		c33		
2.3. Dealing with Problem Complexity		c3s2	c1s1, c20s1-s5			
2.4. Interacting with Stakeholders			*			
2.5. Dealing with Uncertainty and Ambiguity			c4s1, c4s4, c11s5 c24s5			c9s4
2.6. Dealing with Equity, Diversity, and Inclusivity			*			c10s7
3. Communication Skills						
3.1. Reading, Understanding, and Summarizing			c4s5	c33s3		
3.2. Writing		c1s5	c4s2-s3			
3.3. Team and Group Communication		c1s6	c22s3	c27s1		c10s4
3.4. Presentation Skills		c1s5	c22			c10s7-s8

FURTHER READING

Gerald M. Weinberg, *The Psychology of Computer Programming* [15].

This was the first major book to address programming as an individual and team effort; it has become a classic in the field.

Kinney and Lange, P.A., *Intellectual Property Law for Business Lawyers* [16].

This book covers IP laws in the US. It not only talks about what the IP law is; it also explains why it looks the way it does.

REFERENCES

- [1*] F. Bott et al., *Professional Issues in Software Engineering*, 3rd ed., Taylor & Francis, 2000.
- [2] Appendix B of this Guide.
- [3*] G. Voland, *Engineering by Design*, 2nd ed., Prentice-Hall, 2003.
- [4*] I. Sommerville, *Software Engineering*, 10th ed., Addison-Wesley, 2016.
- [5*] S. McConnell, *Code Complete*, 2nd ed., Microsoft Press, 2004.
- [6] 25 Years Washington Accord, IEC, 2014.
- [7] EUR-ACE, 2017.
- [8] ISO/IEC 24773-1, Software and Systems Engineering – Certification of Software and Systems Engineering Professionals — Part 1: General Requirements.
- [9] Software Professional Certification Program IEEE-CS,

https://www.computer.org/education/certification_s.

- [10] ACM Code of Ethics and Professional Conduct, 2018.
- [11] IEEE Code of Ethics, 2020.
- [12] IFIP Code of Ethics and Professional Conduct, 2021.
- [13*] S. Tockey, *Return on Software: Maximizing the Return on Your Software Investment*, Addison-Wesley, 2004.
- [14*] R. E. Fairley, *Managing and Leading Software Projects*, Wiley-IEEE Computer Society Press, 2009.

[15] G. M. Weinberg, *The Psychology of Computer Programming: Silver Anniversary Edition*, Dorset House, 1998.

[16] Kinney and Lange, P.A., *Intellectual Property Law for Business Lawyers*, Thomson West, 2013

CHAPTER 15

SOFTWARE ENGINEERING

ECONOMICS

ACRONYMS

IRR	Internal Rate of Return
MARR	Minimum Acceptable Rate of Return
SDLC	Software Development Life Cycle
SPLC	Software Product Life Cycle
ROI	Return on Investment
SEI	Software Engineering Institute
TCO	Total Cost of Ownership

INTRODUCTION

Software is ubiquitous and has become essential for many organizations. It serves organizations in the following ways:

- as a lever to reach an organization's business or strategic goals;
- as a catalyst of organizational know-how to improve value.

Both aspects lead directly to critical software engineering demands:

- increased productivity;
- long-term sustainability;
- innovation;
- competitiveness;

- alignment with organizational goals.

Software engineering economics helps software engineers work in ways that satisfy these critical demands. The Introduction to *SWEBOk Guide* explains that engineering economics is a key element of all recognized engineering disciplines. Economics is the science of choice, not the science of money. Engineering economics is the applied microeconomics branch of economics. It asks the fundamental question, “Is it in the best interest of this enterprise to invest its limited resources in this technical endeavor, or would the same investment produce a higher return elsewhere?” Paraphrasing the definition in [1], engineering is “finding the balance between what is technically feasible and what is economically acceptable.”

Software engineering must be *value-based*. Neutral — or worse, negative — value from an organization’s investment in software is not sustainable. Software engineering economics aligns software technical decisions with the organization’s business goals.

“The organization” will at least include the organization where the software engineer is employed. However, when the software engineer is involved in work for any third party, such as through an external digital transformation contract or other “works for hire” situation, the business goals of that third party are also relevant.

In all types of organizations — for-profit, nonprofit and government — a value-based approach translates into long-term sustainability. In for-profit organizations this means achieving a tangible return on the software investment. In nonprofit organizations, this means achieving the maximum benefit for the least cost.

Software technical decisions, like an organization's decision to use a preexisting library or to develop its own, may appear easy from a purely technical perspective. However, they can have serious implications for the business viability of a software project as well as the product itself. Most software practitioners wonder whether such concerns apply to them. But economic decision-making is fundamental to engineering. Someone who cannot make decisions from both a technical and an economic perspective cannot be considered a true engineer.

Software engineering economics applies to decisions across the entire software product life cycle (SPLC), from the pre-project decision to develop the software to end-of-life decisions for existing software. It also applies to decisions at all levels of technical detail. For example, all following questions involve an economic perspective:

- can a client organization benefit from a digital transformation?
- does a project proposal (a tender) align with a client's business goals?
- should certain software functionality be bought or built?
- should certain requirements be included in scope or not?
- what is the most efficient, cost-effective architecture and design?

- what is an optimal load-balancing strategy for a cloud-based deployment that provides adequate response time to clients without incurring unnecessary operational cost?
- how much risk-based testing is enough?
- is it better to refactor, redevelop or just live with code that has high technical debt?
- is it better to focus maintenance on adding new functionality or on fixing known defects?
- would the value of early delivery of partial functionality gained by using an Agile process outweigh the overhead of rework and continuous testing inherent in iterative approaches?

The Software Engineering Economics Knowledge Area (KA) is directly or indirectly related to all other KAs in this *Guide*.

This KA also takes the position that the more traditional, purely financial view of engineering economics needs to be broadened [2]. Value does not always derive from money alone; value can also derive from “unquantifiables” like corporate citizenship, employee well-being, environmental friendliness, customer loyalty and so on. Therefore, software engineering decisions must also consider relevant unquantifiable criteria.

BREAKDOWN OF TOPICS FOR SOFTWARE ENGINEERING ECONOMICS

The breakdown of topics for the Software Engineering Economics KA is shown in Figure 15.1.

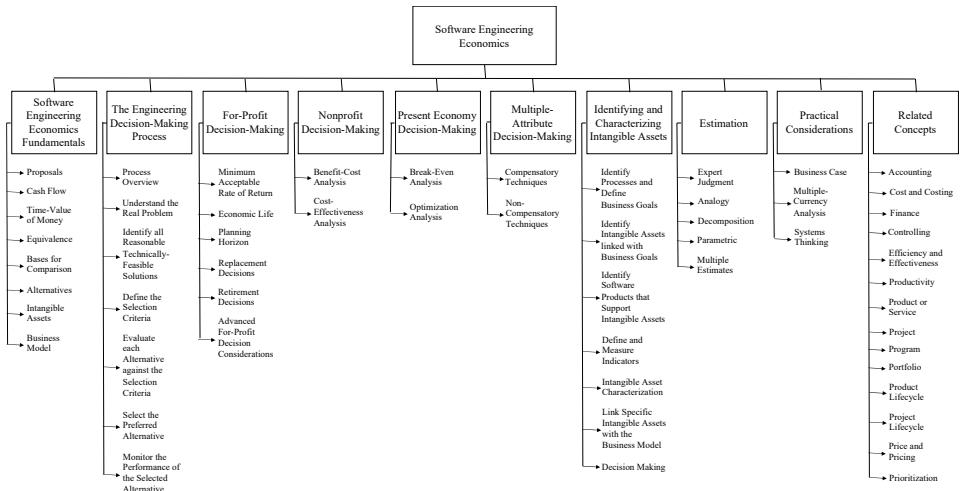


Figure 15.1. Breakdown of Topics for the Software Engineering Economics KA

1. Software Engineering Economics Fundamentals

1.1. Proposals

[3*, c3pp23-24]

Software engineering decisions begin with the concept of a *proposal* — a single, separate course of action to be considered (e.g., carrying out a particular software development project or not). Another proposal could be to enhance an existing software component; another might be to redevelop that same software from scratch. In deciding what algorithm to use in implementing a certain function, each candidate considered is a proposal. Every proposal represents a binary unit of choice — the software engineer either carries out that proposal or chooses not to. Software engineering economics aims to identify the proposals best aligned with the organization's goals.

1.2. Cash Flow

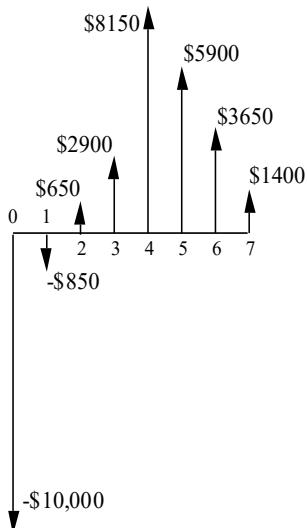
[3*, c3pp24-32]

Engineers must evaluate a proposal from a financial perspective to make a meaningful decision about it. The concepts of *cash flow instance* and *cash flow stream* describe the financial perspective of proposals.

A cash flow instance is a specific amount of money flowing into or out of the organization at a specific time as a direct result of carrying out a proposal. For example, in a proposal to develop and launch product X, the payment for new computers, if needed, would be an example of an outgoing cash flow instance. Money would need to be spent to carry out that proposal. The sales income from product X in the 11th month after market launch would be an example of an incoming cash flow instance. Money would come in because of carrying out the proposal.

A cash flow stream is the set of cash flow instances over time caused by carrying out that proposal. The cash flow stream is that proposal's complete financial view. How much money goes out? When does it go out? How much money comes in? When does it come in? If the cash flow stream for Proposal A is more desirable than the cash flow stream for Proposal B, then — all other things being equal — the organization is financially better off carrying out Proposal A than Proposal B. Thus, the cash flow stream is an important element of engineering decision-making.

A *cash flow diagram* is a picture of a cash flow stream. The cash flow diagram quickly summarizes the financial view of a proposal. Figure 15.2 shows an example cash flow diagram.

**Figure 15.2. A Cash Flow Diagram**

The cash flow stream is shown in two dimensions. Time runs from left to right and amounts of money run up and down. The horizontal axis is divided into units representing years, months, weeks, etc., as appropriate for the proposal. Each net cash flow instance is drawn at a left-to-right position relative to its timing. The amount of the cash flow instance is shown as an upward or downward arrow. Upward arrows indicate that money is coming in (income), whereas downward arrows indicate that money is spent (expense). The arrow's length is usually proportional to the net amount.

1.3. Time-Value of Money

[3*, c5-6]

One of the most fundamental concepts in economics — and therefore, in business decisions — is that money has time-value: Its value changes over time. A specific amount of money right now almost always has a value different from the same amount at some other time. This concept has been around since the earliest recorded human history and is commonly expressed as *interest*.

1.4. Equivalence

[3*, c7]

Due to the time-value of money, two or more cash flows are equivalent only when they equal the same amount of money at the same time. Therefore, comparing cash flows makes sense only when they are expressed in the same time frame. Then, lack of equivalence between the two cash flows can be determined accurately and can serve as the basis for choice. The proposal with the highest value in the same time frame is the most financially desirable.

1.5. Bases for Comparison

[3*, c8]

A *basis for comparison* is a shared frame of reference for comparing two or more cash flow streams. It uses equivalence to meaningfully compare two or more proposals. Several bases for comparison exist, including the following:

- present worth;
- future worth;
- annual equivalent;
- internal rate of return (IRR);
- discounted payback period.

1.6. Alternatives

[3*, c9]

Often, an organization could carry out more than one proposal if it wanted to. But there might be important relationships between proposals that need to be considered. Maybe Proposal Y can only be carried out if Proposal X is also carried out. Or maybe Proposal P cannot be carried out if Proposal Q is carried out, nor could Q be carried out if P were. Decisions are easier when there are mutually exclusive paths — A, or B, or C, or another project, and no others. This topic explains how to turn any set of proposals, with their interrelationships, into a set of mutually exclusive alternatives. The cash flow stream for any alternative is the sum of the cash flow streams for all the proposals it contains. The choice can then be made among these alternatives.

One special case is known as the *do-nothing alternative*. Sometimes the best course of action is not to carry out any of the proposals being considered. The do-nothing alternative represents that case. It doesn't mean do nothing at all; it means

“do something else, something that’s not in this set of choices.” The do-nothing alternative should be considered in most, but not all, situations.

1.7. Intangible Assets

Intangible assets, also known as *knowledge assets*, are any knowledge that lies in the non-visible side of an organization but affects that organization’s financial performance. According to International Valuation Standards (IVS) 210 § 20.1, “an intangible asset is a non-monetary asset that manifests itself by its economic properties. It does not have physical substance but grants rights and economic benefits to its owner” [4].

This can include, but is not limited to, policies, procedures, tools and specifications, as well as organizational culture, experience and know-how.

Knowing the organization’s intangible assets helps the software engineer better understand how proposals may affect or be affected by organizational realities. Otherwise, hidden risks and opportunities that could influence proposals’ success or failure might not be exposed.

The skills needed to consider intangible assets are the following:

- intangible assets identification and valuation [Skills Framework for the Information Age (SFIA), category Strategy and Architecture, subcategory Business strategy and planning];
- knowledge management [SFIA, category Strategy and Architecture, subcategory Business strategy and planning].

Identifying and characterizing intangible assets are discussed in more detail later in this KA.

1.8. Business Model

Peter Drucker, a founder of modern management, defines a good business model as one that answers these questions [5]:

- “Who is the customer?”
- “What does the customer value?”
- “How do we make money?”
- “What is the underlying economic logic that explains how we can deliver value to customers at an appropriate cost?”

Understanding the organization’s business model — as well as its intangible assets — helps the software engineer better understand how proposals may affect or be affected by organizational realities. Analyzing the business model can help the software engineer identify hidden risks and opportunities that could influence a proposal’s success or failure [6].

The skills needed to consider intangible assets are knowledge management and intangible assets identification and valuation [SFIA, category Strategy and Architecture, subcategory Business strategy and planning].

2. The Engineering Decision-Making Process

2.1. Process Overview

[3*, c4pp35-36]

Figure 15.3 provides an overview of the engineering decision-making process.

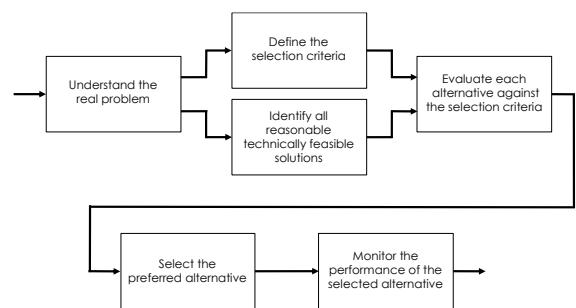


Figure 15.3. The Engineering Decision-Making Process

The process is shown as stepwise and sequential; however, it can be more fluid in practice. Steps can be done iteratively, can overlap and can even occur in different sequences. Just be sure not to skip any step or execute it poorly.

When the consequences of a wrong decision are significant, such as a go/no-go decision for a large project, more time, effort and care should be spent in this process. All steps should be completed thoroughly and carefully. ISO 12207 [7] and ISO

15288 [8] recommend two additional early activities, which can be important in high-consequence decisions:

- define the decision management strategy — this strategy might specify roles, responsibilities, procedures and tools;
- identify relevant stakeholders, which might include appropriate subject matter experts.

When the consequences of a wrong decision are small, such as the consequences of selecting a minor algorithm or data structure, less time, effort and care can be spent, but the same general process is followed. Each step is discussed in more detail below.

2.2. Understand the Real Problem

[3*, c4pp37-39]

The best solution to a problem can come only from thoroughly understanding the real problem to be solved. This step's key aspects include the use of an interrogative technique such as the “5 Whys” technique and a consideration of the broader context surrounding the problem. The Empathize stage in Design Thinking [9] (to consider intangible assets) and looking closely at the organization's business model are examples of considering that broader context.

2.3. Identify All Reasonable Technically Feasible Solutions

[3*, c4pp40-41]

The goal of engineering decision-making is to find the best solution. However, the best solution must first be identified as a candidate before it can be selected as the best. If the best solution is not among the set of solutions being considered, it cannot be selected. The importance of creative thinking in this step cannot be overstated when the consequences of a wrong decision are high.

For some potential solutions, or candidates, prototyping is a useful way to verify technical feasibility. Peer review can also help verify technical feasibility and possibly spur the identification of even more candidates.

On the one hand, adding candidates increases the probability that the best one is in the set. On the other hand, each adds cost to the decision-making process. Software engineers must use their best judgment in deciding when they have enough candidates. These candidates are the “proposals,” as defined in the Fundamentals topic.

2.4. Define the Selection Criteria

[3*, c4pp39-40, c26pp441-442]

Engineering decisions almost always consider the financial perspective. However, other decision criteria can also be relevant; when this is the case, the decision is a multiple-attribute decision. For example, an environmentally conscious organization may choose a less economical solution if it is more eco-friendly. In many cases, the greater the consequences of a wrong decision, the more selection criteria need to be considered.

As much as possible, each criterion should be expressed objectively. Ideally, those objective terms will be expressed as a monetary value — but not necessarily. What is the “value” of a clean river? It might not make sense to value a river by multiplying the price per kilogram of fish by an estimate of the number of fish in the river. Decision criteria that can't be expressed objectively are called “unquantifiables,” “irreducibles” or “intangibles.”

Defining the decision criteria can be a subjective task. Too many criteria could make the analysis unwieldy. On the other hand, too few criteria might not differentiate well between proposals and could thus lead to a suboptimal choice. The potential for a better decision provided by including more criteria must be balanced against the extra effort required to analyze the criteria.

To the extent that money is a selection criterion, the context of the decision will constrain the decision-maker to a for-profit, nonprofit or present economy decision analysis, as explained in topics 3, 4 and 5, later in this KA.

2.5. Evaluate Each Alternative Against the Selection Criteria

[3*, c4pp41-42]

Each alternative is evaluated against each selection criterion. When a selection criterion involves money, each alternative must be judged from the same viewpoint. Use the same basis for comparison (present worth, future worth, IRR, etc., in for-profit decisions; benefit-cost ratio or cost-effectiveness in nonprofit decisions, etc.), the same planning horizon, and consider the same kinds of costs and incomes. An example decision might be buying and adapting an off-the-shelf software product versus building a custom application from scratch. Considering costs for a longer time frame for one proposal than for the other will make the one using the shorter time frame seem like the better choice even though it might not be better over the same time frame.

2.6. Select the Preferred Alternative

[3*, c4p42, c26pp447-458]

If the only selection criterion is money, the alternative with the highest present worth, future worth, etc., will be chosen. When there are multiple criteria, a variety of techniques can be used to evaluate the criteria together. Multiple-attribute decision-making is detailed later in this KA.

Engineering decisions are based on estimates (discussed later in this KA). The accuracy of an estimate is limited in theory and in practice, and the degree of inaccuracy depends on the specifics of the situation [3*, c21pp344-356]. If the degree of inaccuracy is high enough, that inaccuracy could change the resulting decision. The following techniques [3*, c23] can help engineers address these situations:

- consider ranges of estimates;
- perform a sensitivity analysis;
- delay final decisions.

In addition, two categories of techniques address multiple potential outcomes from a decision:

- decision-making-under-risk techniques [3*, c24] are used when probabilities can be assigned to the different potential outcomes. Specific techniques include expected value decision-making, expectation variance and decision-making, Monte Carlo analysis,

decision trees, and the expected value of perfect information;

- decision-making-under-uncertainty techniques [3*, c25] are used when probabilities cannot be assigned to the different potential outcomes. Specific techniques include the Laplace Rule, the Maximin Rule, the Maximax Rule, the Hurwicz Rule and the Minimax Regret Rule.

High-consequence decisions may benefit from formally recording the selected alternative and the justification for why that alternative was selected.

2.7. Monitor the Performance of the Selected Alternative

[3*, c4pp42-43]

Because estimation is a fundamental element of engineering decision-making, the quality of the decision depends on the quality of the estimates. Bad estimates can easily lead to bad decisions. The software engineer needs to “close the loop” on estimates by comparing them to the actual outcomes. Otherwise, no one will ever know if the estimates were good [3*, c21pp356-358]. This also helps improve estimation over time. Understanding what drives differences between estimates and actual outcomes helps engineers refine estimation techniques to produce more accurate estimates in the future.

3. For-Profit Decision-Making

For-profit decision techniques apply when the organization’s goal is profit — which is the case in most companies.

Figure 15.4 shows the process for identifying the financially best alternative out of a set of proposals. Arranging alternatives in order of increasing initial investment and then selecting strictly better candidates means that, all other considerations being equal, the alternative with the smaller initial investment will be chosen. The “Is the next candidate strictly better?” decision is made in terms of the appropriate basis for comparison: present worth, future worth, IRR, etc.

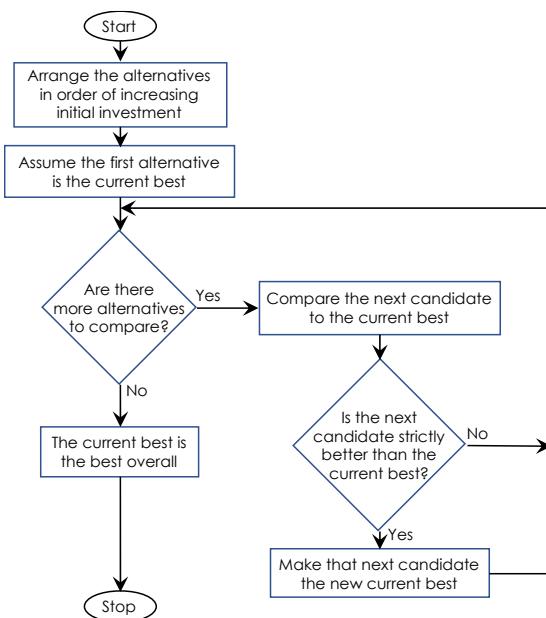


Figure 15.4. The For-Profit Decision-Making Process

3.1. Minimum Acceptable Rate of Return

[3*, c10pp141-143]

The *minimum acceptable rate of return* (MARR) is the lowest IRR the organization would consider a good investment. Generally, it would not be wise to invest in an activity with a return of 10% when another activity returns 20%. The MARR is a statement that the organization is confident it can achieve at least that rate of return. The MARR represents the organization's opportunity cost for investments. By investing in some alternative, the organization explicitly decides not to invest that same money somewhere else. If the organization is already confident it can achieve a known rate of return, alternatives should be chosen only if their rate of return is at least that high. A simple way to account for that opportunity cost is to use the MARR as the interest rate in the basis for comparison.

3.2. Economic Life

[3*, c11pp160-164]

When an organization invests in a particular alternative, money is tied up in that alternative — so-called *frozen assets*. The economic impact of frozen assets typically starts high and decreases over

time. On the other hand, operating and maintenance costs tend to start low and increase over time. An alternative's total cost is the sum of those two costs. At first, frozen asset costs dominate; later, operating and maintenance costs dominate. At some point, the sum of the two costs is minimized; this is the *economic life* or *minimum cost lifetime*.

3.3. Planning Horizon

[3*, c11]

To properly compare a proposal with a four-year life to a proposal with a six-year life, the economic effects of either cutting the six-year proposal by two years or investing the profits from the four-year proposal for another two years need to be addressed. The *planning horizon*, sometimes known as the *study period*, is the consistent time frame over which all proposals in the same decision are considered. Aspects such as economic life and the time frame over which reasonable estimates can be made need to be factored into establishing a planning horizon. Once the planning horizon is established, several techniques are available for putting proposals with different life spans into that planning horizon.

3.4. Replacement Decisions

[3*, c12pp171-178] [8*, c9]

A replacement decision happens when an organization already has a particular asset and is considering replacing it with a different asset (e.g., deciding between maintaining and supporting legacy software or redeveloping it from the ground up). Replacement decisions use the same for-profit decision process, but there are two additional important considerations: sunk cost and salvage value.

3.5. Retirement Decisions

[3*, c12pp178-181] [8*, c9]

Retirement decisions are about getting out of an activity altogether, such as when a software company considers not selling a software product anymore or a hardware manufacturer considers not building and selling a particular computer model any

longer. Retirement decisions can be preplanned or happen spontaneously (e.g., when performance targets are not achieved). Retirement decisions can be influenced by lock-in factors such as technology dependency and high exit costs.

3.6. Advanced For-Profit Decision Considerations

[3*, c13-17]

The above concepts and techniques are often sufficient to make a good for-profit decision. However, particularly when the consequences of a wrong decision are high, additional considerations may need to be factored into the decision analysis, including the following:

- inflation or deflation;
- depreciation;
- income taxes.

4. Nonprofit Decision-Making

The for-profit decision techniques don't apply when the organization's goal isn't profit — which is the case in government and nonprofit organizations. These organizations have a different goal, so different decision techniques are needed. The two techniques are benefit-cost analysis and cost-effectiveness analysis (discussed below).

4.1. Benefit-Cost Analysis

[3*, c18pp303-311]

Benefit-cost analysis is one of the most widely used methods for evaluating proposals in nonprofit organizations. A proposal's financial benefits are divided by its costs. Any proposal with a benefit-cost ratio of less than 1.0 can usually be rejected without further analysis because it would cost more than it would benefit the organization. Additional considerations are necessary when two or more proposals are considered at the same time.

4.2. Cost-Effectiveness Analysis

[3*, c18pp311-314]

Cost-effectiveness analysis shares much of the philosophy and methodology of benefit-cost analysis. There are two versions of cost-

effectiveness analysis. The fixed-cost version seeks to maximize benefit given a fixed upper bound on cost. The fixed-effectiveness version seeks to minimize the cost to achieve a fixed goal.

5. Present Economy Decision-Making

This subset of engineering decision-making is called *present economy* because it does not involve the time-value of money (future economy). The two forms of present economy decisions are presented below.

5.1. Break-Even Analysis

[3*, c19]

Given functions describing the costs of two or more proposals, break-even analysis helps engineers choose between them by identifying points where those cost functions are equal. Below a break-even point, one proposal is preferred, and above that point, the other is preferred. For example, consider a choice between two cloud service providers. One provider has a lower fixed cost per month with a higher incremental fee for use, whereas the other has a higher fixed monthly cost with a lower incremental fee for use. Break-even analysis identifies the use level where the costs are the same. The organization's expected use level can be compared to the break-even point to identify the lower-cost provider.

5.2. Optimization Analysis

[3*, c20]

Optimization analysis studies one or more cost functions over a range of values to find the point where overall cost is lowest. Software's classic space-time trade-off is an example of optimization; an algorithm that runs faster often uses more memory. Optimization balances the value of faster run time against the cost of the additional memory.

6. Multiple-Attribute Decision-Making

[3*, c26]

Most topics presented in this KA so far have discussed decisions based on a single criterion — money. The alternative with the best present worth, the best incremental IRR, the best incremental benefit-cost ratio, etc., is the one selected. Aside from technical feasibility, money is usually the most important decision criterion, but it's certainly not always the only one. Often, other criteria, other "attributes," need to be considered that can't be cast in terms of money. Multiple-attribute decision-making techniques allow nonmonetary criteria to be factored into the decision.

A variety of techniques can be used to address multiple criteria, including nonmonetary criteria. These techniques fall into two categories.

6.1. Compensatory Techniques

[3*, c26pp449-458]

Also called *single-dimensioned techniques*, the techniques in this category collapse all criteria into a single figure of merit. This category is called *compensatory* because, for any given alternative, a lower score in one criterion can be compensated by — traded off against — a higher score in other criteria. Compensatory techniques include nondimensional scaling, additive weighting and analytic hierarchy process (AHP).

Gilb's Impact Estimation [11] and the Software Engineering Institute's (SEI) Architectural Tradeoff Analysis Method (ATAM) [12] are examples of compensatory multiple-attribute decision-making techniques focused on identifying the best software design.

6.2. Non-Compensatory Techniques

[3*, c26pp447-449]

Also called *fully dimensioned techniques*, the techniques in this category do not allow trade-offs among the criteria. Each criterion is treated as a separate entity in the selection process. Non-compensatory techniques include dominance, satisficing and lexicography.

7. Identifying and Characterizing Intangible Assets

The intangible side of an organization is the valuable knowledge residing within it. This includes employees' knowledge about processes, structures, procedures, etc. (tacit, or implicit, knowledge), as well as institutional knowledge recorded in various organizational resources (explicit knowledge). These assets are usually hidden, the way most of an iceberg is underwater. The intangible assets must be explicitly considered in many decisions, particularly when the consequences of a wrong decision are high for the client, no matter if the client is internal or external to the organization for which the software project is being done.

If these assets are not adequately considered, software engineers risk developing a software solution that does not fit the client organization. Only when the intangible assets are explicitly considered will the risk of a poorly fitting software solution be minimized. The Strategic Intangible Process Assets Characterization (SIPAC) method [13] has been used to good effect to accomplish this. SIPAC steps are outlined in the following subsections.

7.1. Identify Processes and Define Business Goals

Start by understanding the organization's business processes and business goals. If the organization has well-documented processes, these should be used; otherwise, a deliberate survey will be necessary.

Business goals can include, but are not limited to, the following:

- maintaining growth and continuity of the organization;
- meeting financial objectives;
- meeting responsibility to employees;
- meeting responsibility to society;
- managing market position.

7.2. Identify Intangible Assets Linked with Business Goals

The next step is to comprehensively identify the intangible assets. Common examples are policies, documented business processes, checklists, lessons learned, templates, standards, procedures, plans and training materials. Organizations develop or acquire these assets to meet their business goals. The assets represent investments that provide business value. One effective approach to identifying an organization's intangible assets is to start with a taxonomy such as described in the following reference [14]. The goal is to identify as many intangible assets as possible that serve as a lever to achieve the business goals identified in the previous step. In practice, this can be an iterative process where reviewing the already-identified assets reveals the existence of others. A practical way to do it is by focusing iteratively on the 11 generic intangible assets (GIAs) described in [6].

Possible GIAs represent all potential parts of any business that can be involved in a digital transformation. Focusing on one or more of them allows the software engineer to better understand and frame the project's impact. Focusing iteratively on the 11 GIAs, the software engineer will select the type of GIA to be considered and, with this, elicit the specific intangible assets associated with each GIA. In addition to identifying specific intangible assets, a qualitative relative "importance" must be added to each one as it is identified. The importance is a value between 1 and 5 (1 for lower importance and 5 for higher importance), depending on how well the asset supports achieving the business objectives. The intangible assets with the highest importance are likely the most suitable target for the client organization.

7.3. Identify Software Products That Support Intangible Assets

Software products that support specific intangible assets will be part of the digital transformation proposal to be presented to the client to help them decide what digital transformation to implement.

To identify products related to specific intangible assets, the software engineer can choose from the following:

- discovering them all at a single time by using the methodology of Osterwalder [13], which promotes innovation by generating a value map with the stakeholders' emerging needs, mapping the products to the specific intangible assets;
- listing them if they are known and then mapping them to specific intangible assets;
- iteratively working with the individual intangible assets by (1) selecting a specific intangible asset and (2) identifying the products, continuing until all specific intangible assets have been analyzed.

A single product can support more than one specific intangible asset, and each specific intangible asset can be supported by many products.

7.4. Define and Measure Indicators

This step defines and measures the indicators that will be used to characterize how the intangible assets (through the software products that support those intangible assets) help meet business goals through describing, implementing or improving the identified products. Quality indicators assess specific intangible asset characteristics or features. Impact indicators assess how much the specific intangible assets contribute to processes or business goals.

Indicators must be normalized and standardized to operate correctly.

7.5. Intangible Asset Characterization

Based on the information gathered, the software engineer determines the value of the identified specific intangible assets based on their quality and impact. Specific intangible assets may be characterized in terms of their impact on business goals and their quality as organizational assets. There are three important characterization cases:

- case 1: specific intangible assets with both impact and quality indicators (Warning, Replaceable, Evolving or Stable);

- case 2: specific intangible assets with only quality indicators (Acceptable or Unacceptable);
- case 3: specific intangible assets with only impact indicators (Acceptable or Unacceptable).

The three characterization cases are shown in Figure 15.5. The quadrants represent the “states” constituting different levels of characterization. The lines separating the quadrants are thresholds of impact and quality that define the point at which the impact or quality of a specific intangible asset may be considered acceptable or not for each organization. These thresholds are established for every client organization and specify what level of organizational performance, quality, and impact they will demand from their knowledge/intangible assets.

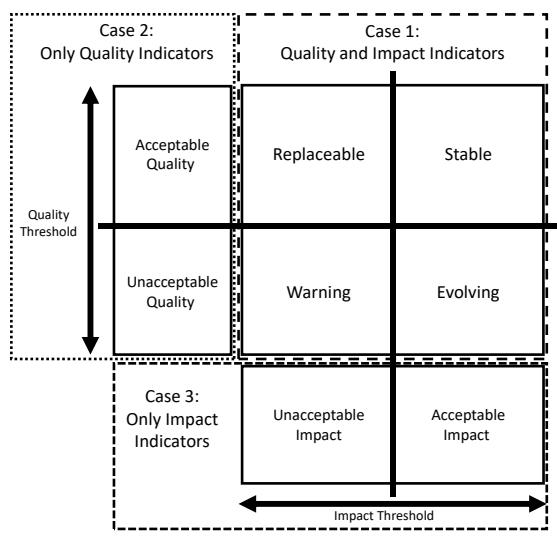


Figure 15.5. Extended Characterization of Specific Intangible Assets

The characterization uses information from standardized-normalized indicators to assess the identified intangible assets. This assessment generates a descriptive value that will determine the asset’s general state of health from a quantitative perspective.

Quality quantitative assessment

The quality valuation considers only the indicators of the type ***quality*** of an intangible asset and calculates a general valuation of it. To evaluate the subset of quality indicators, given a set of q quality indicators for an intangible asset n , the valuation of the quality is given according to Equation 1.

Equation 1. Quality Assessment for a Knowledge Asset

$$Q_{VAL}^n = \frac{\sum_{i=1}^q X_i^n}{q}$$

Where X_i^n is each of the q normalized indicators of quality that the intangible asset n has.

If n is the number of indicators of quality and n is the number of indicators of impact, the assessments of quality and impact are given as the average of the normalized values of the corresponding indicators.

$$Qval = \frac{\sum_{i=1}^n X_{NORM}^i}{n}$$

Above, $Qval$ is the average of the normalized values of the quality indicators of a corresponding specific intangible asset.

Impact quantitative assessment

An intangible asset’s impact valuation is an assessment that considers only the normalized indicators that are classified as “impact” indicators. To evaluate the subset of impact indicators, given a set of p normalized impact indicators for an intangible asset n , the valuation is given as stated in Equation 2:

Equation 2. Impact Assessment for a Specific Intangible Asset

$$I_{VAL}^n = \frac{\sum_{i=1}^p X_i^n}{p}$$

Where X_i^n is each of the p normalized indicators of impact that the knowledge asset n has.

$$Ival = \frac{\sum_{i=1}^m X_{NORM}^i}{n}$$

Where $Ival$ is the average of the normalized values of the impact indicators of a corresponding knowledge asset.

Finally, the linear value of an intangible asset characterization is given by the impact and quality valuations (*Qval* and *Ival*), following these rules, assuming that both quality and impact are equally important, so *KAval* (the valuation of the intangible asset) is given by:

$$\text{If } \exists Qval \cap \exists Ival, \text{ then } KAval = \frac{Qval+Ival}{2}$$

$$\text{If } \exists Qval \cap \nexists Ival, \text{ then } KAval = Qval$$

$$\text{If } \nexists Qval \cap \exists Ival, \text{ then } KAval = Ival$$

This linear value represents an intangible asset's general state based on the state of its indicators. It uses the algebraic mean of the standardized and normalized indicators to represent the assets' general state on a scale [-1, 1] and based on the corresponding interpretation thresholds.

7.6. Link Specific Intangible Assets with the Business Model

Visualizing the client business model, enriched with the intangible assets status allocated into that model, gives organizational leadership a clear understanding of the important relationships among proposed software solutions, intangible assets, the business model and the business goals. The software engineer can clearly show which proposed solution generates the most value for the business. An example is shown in [6].

7.7. Decision-Making

The next step in the decision-making process is to prioritize and choose the software products that interest the client organization most. There is no simple rule; several criteria must be considered:

- the intangible asset's impact on business goals (defined in previous steps);
- the characterization reached (defined in previous steps);

- the impact of intangibles assets status on the competitors of the organization under improvement;
- the intangible asset's impact on the business model;
- cost to implement the products;
- time to implement the products;
- complexity of the products.

All criteria must be considered to decide what software products should be developed for the client organization, making this a multiple-attribute decision. (See 6., Multiple-Attribute Decision-Making.)

Upon considering all relevant criteria, the organization can see the risks of implementing a software solution to automate processes that are either not very valuable or not in good shape. Instead, the software engineer can offer, in a transparent way, proposals that better satisfy the organization's business needs.

This approach can be useful whenever an engineering decision needs to be made, but it is particularly critical in the pre-project stage when there is a need to present the client organization with proposals that are best for business value.

8. Estimation

[3*, c21-26]

An estimate analytically predicts some quantity, like a project's size, cost or schedule. Many other quantities are also estimated in software engineering, such as the average number of active client sessions a cloud service needs to support, the number of times a function will be called during execution of a section of code, or the number of delivered defects in a software product.

Software engineers do not estimate purely for the sake of estimating. Software engineers estimate to make decisions when critical quantities are unknown. For example, a decision to buy a functionality or build it within the organization would certainly be based on the cost of building it. But the actual cost of building it cannot be precisely

known until the organization does build it. Key information needed to make engineering decisions is usually not known when those decisions need to be made. Instead, decisions are based on estimates. Behind every estimate is one or more decisions.

Given that estimates are predictions, there is a nonzero probability that the actual outcome will differ from the estimate. All estimates are inherently uncertain. Sometimes, the uncertainty is large, and sometimes it is small. But it is always there. Fortunately, estimates need not be perfect; they need only to be good enough to lead the decision-maker to make the right decision.

The Software Engineering Code of Ethics and Professional Practice [16] states, “3.09. Ensure realistic quantitative estimates of cost, scheduling, personnel, quality and outcomes on any project on which they work or propose to work and provide an uncertainty assessment of these estimates” (underlining added for emphasis). (See [3*, c21pp358-361].)

Estimation is covered extensively in [17], [18] and [3*]. Several general techniques exist, and each is overviewed here. All specific estimation techniques use one or a combination of these general techniques.

8.1. Expert Judgment

[3*, c22pp367-369]

In expert judgment estimation, the estimate is based purely on the estimator’s professional opinion. It is the simplest technique and is always available, and it is particularly useful when the other techniques aren’t available. The downside is that this technique produces the least accurate estimates. Multiple expert judgment estimates can be fed into group estimation processes like Wide Band Delphi and Planning Poker to produce more accurate estimates.

8.2. Analogy

[3*, c22pp369-371]

Estimation by analogy assumes that if the thing estimated is similar to something already known, then the estimate for the new thing can be based on the actual result for the similar thing, with

allowances for relevant differences. The steps in estimation by analogy are as follows:

1. Understand the thing to be estimated.
2. Find a suitable analogy for which actual results are known.
3. List differences between the thing being estimated and the analogy that could significantly affect the outcome.
4. Estimate the magnitude of each identified difference.
5. Build the estimate from the analogy’s actual result and adjustments for the identified differences.

Estimation by analogy produces more accurate results than expert judgment, and it is still relatively quick and easy. On the other hand, an appropriate analogy for which accurate results are known must be available for this approach to work.

8.3. Decomposition

[3*, c22pp371-374]

Sometimes called *bottom-up estimation*, the steps in estimation by decomposition are:

1. Break the thing to be estimated into successively smaller pieces until the smallest pieces can be reasonably estimated.
2. Estimate those smallest pieces.
3. Add up the estimates for the smallest pieces to build the estimate for the whole.
4. If the estimates for the smallest pieces don’t include allowances for significant cross-cutting factors, then find a way to address those factors. For example, when estimating a software project from its design, the estimates for the design elements may not include allowances for requirements work, integration work, testing work and user documentation work.

Estimation by decomposition assumes that overestimates of lowest-level pieces will cancel out corresponding underestimates of other pieces and lead to a more accurate estimate of the whole. The primary disadvantages are the following:

- it can be a lot more work than any other technique;
- if the bottom-level estimates are biased either high or low, the canceling effect doesn’t happen.

8.4. Parametric

[3*, c22pp374-377]

Also called *estimation by statistical methods*, parametric estimation takes advantage of a known, mathematical relationship between the thing being estimated and one or more observable factors about that thing, like calculating the cost to build a building as a function of its floor space. The estimation model is an equation: First, count the observable factors, and then plug them into the equation to get the resulting estimate.

Parametric estimates are typically the most accurate, the most defendable and the easiest to use, provided the equation has been developed and validated. The disadvantage is that developing and validating such an equation requires an adequate base of accurate historical data along with some nontrivial mathematics and statistics.

8.5. Multiple Estimates

[3*, c22pp377-379]

When the consequences of a wrong decision are small, it can be acceptable to base the decision on a single estimate from a single estimator using a single estimation technique. However, when the consequences of a wrong decision are significant, investing extra effort in developing more than one estimate can be worthwhile.

To use this approach, engineers estimate the same thing using different techniques, possibly by different estimators. Then, they look for convergence or divergence among those multiple estimates. Convergence suggests the individual estimates are probably accurate, and any of them could be used to make the decision. Divergence suggests that one or more important factors might have been overlooked. Finding the factors that caused the divergence and reestimating to produce converging results often lead to a better estimate and thus a better decision.

9. Practical Considerations

9.1. Business Case

The business case is the consolidated, documented information summarizing and explaining a recommended business decision from different perspectives (cost, benefit, risk and so on) for a decision-maker and other relevant stakeholders. It's used to assess a product's potential value, which can be used as a basis for an investment decision.

9.2. Multiple-Currency Analysis

When a decision analysis involves cross-border finances, currency exchange rate variations may need to be considered. This is often done using historical data.

9.3. Systems Thinking

The ecosystem in which software engineers develop their professional life is complex. To understand the whole picture around a client organization and form a holistic view of the scenarios they analyze, software engineers can use systems thinking methodologies. This approach helps the software engineer create a complete set of possible scenarios in which the software to be provided could be useful and, with this information, explain to the client how the software solution can be a value provider for the organization. Sources for system thinking methodologies are [19] and [20]. A way to connect systems thinking methodologies with the development of a business model to understand the pillars of the client organization can be reached here [21].

10. Related Concepts

This topic includes concepts the software engineer may want to bear in mind.

10.1. Accounting

[3*, c15pp234-245]

Accounting is part of finance. It allows people whose money is used to run an organization to know

the results of their investment: Did they get the profit they were expecting? In for-profit organizations, this relates to the tangible return on investment (ROI), while in nonprofit and governmental organizations, as well as for-profit organizations, it translates into sustainably staying in business. Accounting's primary role is to measure the organization's actual financial performance and to communicate financial information about a business entity to stakeholders, such as shareholders, financial auditors and investors. Communication generally takes the form of financial statements showing the economic resources to be controlled. The right information — relevant and reliable to the user — must be presented. Information and its timing are partially governed by risk management and governance policies. Accounting systems are also a rich source of historical data for estimating.

Software engineers must be conscious of the software's importance as a driver of business accounts in the digital era.

10.2. Cost and Costing

[3*, c15pp245-259]

A cost is the money used to produce something and, hence, is no longer available for use. In economics, a cost is an alternative that is given up as a result of a decision.

Sunk cost refers to unrecoverable expenses that have occurred, which can cause emotional hurdles looking forward. From a traditional economics viewpoint, sunk costs should not be considered in decision-making. Opportunity cost is the cost of an alternative that must be forgone to pursue another alternative.

Costing is part of finance and product management. It is the process of determining the cost based on expenses (e.g., production, software engineering, distribution, rework) and on the target cost to be competitive and successful in a market. The target cost can be below the actual estimated cost. The planning and controlling of these costs (called *cost management*) is important and should always be included in costing.

An important concept in costing is the total cost of ownership (TCO). This holds true especially for software because there are many not-so-obvious costs related to SPLC activities after initial product development. TCO for a software product is defined as the total cost for acquiring that product, activating it and keeping it running. These costs can be grouped as direct and indirect costs. TCO is an accounting method that is crucial in making sound economic decisions.

10.3. Finance

Finance is the branch of economics concerned with allocating, managing, acquiring and investing resources. Finance is an element of every organization, including software engineering organizations.

The field of finance deals with the concepts of time, money, and risk, and how they are interrelated. It also deals with how money is spent and budgeted. Corporate finance is concerned with funding an organization's activities. Generally, this involves balancing risk and profitability while attempting to maximize an organization's wealth and the value of its stock. This applies primarily to for-profit organizations but also to nonprofit organizations. The latter needs finances to ensure sustainability, if not to make a tangible profit. To do this, an organization must:

- identify organizational goals, time horizons, risk factors, tax considerations and financial constraints;
- identify and implement the appropriate business strategy, such as which portfolio and investment decisions to take, how to manage cash flow and where to get the funding;
- measure financial performance, such as cash flow and ROI, and take corrective actions in case of deviation from objectives and strategy.

Provided that many organizations use software development or acquisition to stay competitive, the software engineer must be conscious of the importance of software to business finances.

10.4. Controlling

Controlling is the element of finance and accounting that involves measuring and correcting performance. It ensures that an organization's objectives and plans are accomplished. Controlling cost is a specialized branch of controlling used to detect variances of actual costs from planned costs.

In software engineering, this concept is referred to as processes and products control and evolution. While the organization is seen as an entity with its own goals, and control of the organizational goals is seen as separate, software engineers must consider control of the organization part of their job by ensuring alignment of their software with business goals.

10.5. Efficiency and Effectiveness

[10*, c22pp422-23]

Economic efficiency of a process, activity or task is the ratio of resources consumed to resources expected to be consumed. Efficiency means “doing things right.” An efficient behavior, like an effective behavior, delivers results and minimizes effort. Factors affecting efficiency in software engineering include product complexity, quality requirements, time pressure, process capability, team distribution, interruptions, feature churn, tools and programming language.

Effectiveness is about having impact. It is the relationship between achieved objectives and defined objectives. *Effectiveness* means “doing the right things.” Effectiveness looks only at whether defined objectives are reached — not at how they are reached.

10.6. Productivity

[10*, c23pp689]

Productivity is the ratio of output to input from an economic perspective. *Output* is the value delivered. *Input* covers all resources (e.g., effort) spent to generate the output. Productivity combines efficiency and effectiveness from a value-oriented perspective. Maximizing productivity is about generating the highest value with the lowest resource consumption.

10.7. Product or Service

A *product* is a tangible economic good (or output) created in a process that transforms product factors (or inputs) into an output. A *service* is an intangible resource, like consulting. When sold, a product or service is a deliverable that creates both a value and an experience for its consumers. A product or service can be a combination of systems, solutions and materials delivered internally (e.g., an in-house IT solution) or externally (e.g., a software application), either as is or as a component for another product (e.g., embedded software).

10.8. Project

[22*, c2s2.4]

A *project* is “a temporary endeavor undertaken to create a unique product, service, or result” [23]. In software engineering, different project types are distinguished (e.g., product development, outsourced services, software maintenance, service creation, and so on). During its life cycle, a software product may require many projects. For example, during the product conception phase, a project might be conducted to determine customer need and market requirements; during maintenance, a project might be conducted to produce the next version of a product.

10.9. Program

A *program* is “a group of related projects, subprograms, and program activities managed in a coordinated way to obtain benefits not available from managing them individually” [23]. Programs are often used to identify and manage different deliveries to a single customer or market over a time horizon of several years.

10.10. Portfolio

Portfolios are “projects, programs, sub-portfolios, and operations managed as a group to achieve strategic objectives” [23]. Portfolios are used to group and then manage simultaneously all assets within a business line or organization. Having an entire portfolio to consider helps ensure that the broader impacts of decisions are considered, such as the decision to allocate resources to a specific

project, which means that the same resources will not be available for the other projects in the portfolio.

10.11. Product Life Cycle

An SPLC includes all activities needed to define, build, operate, maintain and retire a software product or service and its variants. The SPLC activities of “operate,” “maintain” and “retire” occur in a much longer time frame than initial software development (the software development life cycle (SDLC)). (See Software Life Cycle Models in the Software Engineering Process KA.) Also, the operate-maintain-retire activities of an SPLC consume more total effort and other resources than the SDLC activities. (See Majority of Maintenance Costs in the Software Maintenance KA.) The value contributed by a software product or associated services can be objectively determined during the “operate and maintain” time frame. Software engineering economics should be concerned with all SPLC activities, including activities that take place after the initial product release.

10.12. Project Life Cycle

Project life cycle activities typically involve five process groups: Initiating, Planning, Executing, Monitoring and controlling, and Closing [24]. (See the Software Engineering Management KA.) The activities within a software project life cycle are often interleaved, overlapped and iterated in various ways [20*, c2] [25]. (See the Software Engineering Process KA.) For instance, Agile product development within an SPLC involves multiple iterations that produce increments of deliverable software. An SPLC should include risk management and synchronization with different suppliers (if any) while providing auditable decision-making information (e.g., to comply with product liability needs or governance regulations). The software project life cycle and the SPLC are interrelated; an SPLC may include several SDLCs.

10.13. Price and Pricing

[10*, c23s23.1]

A *price* is what is paid in exchange for a good or service. Price is a fundamental aspect of financial modeling and is one of the four Ps of the marketing mix. The other three Ps are product, promotion and place. Price is the only revenue-generating element among the four Ps; the rest are costs.

Pricing is an element of finance and marketing. It determines what a company will receive in exchange for its products. Pricing factors include manufacturing cost, market placement, competition, market condition and product quality. Pricing applies prices to products and services based on factors such as fixed amount, quantity break, promotion or sales campaign, specific vendor quote, shipment or invoice date, combination of multiple orders, service offerings, and many others. The consumer’s needs can be converted into demand only if the consumer has the willingness and capacity to buy the product. Thus, pricing is crucial in marketing. Pricing is initially done during the project initiation phase and is a part of the “go” decision-making process.

10.14. Prioritization

Prioritization involves ranking alternatives based on common criteria to deliver the best value. For example, in software engineering projects, software requirements are often prioritized to deliver the most value to the client within the constraints of schedule, budget, resources, and technology, or to allow the team to build the product in increments, where the first increments provide the highest value to the customer. (See Requirements Prioritization in the Software Requirements KA and Software Life Cycle Models in the Software Engineering Process KA.) Prioritizing alternatives is at least implicit in the discussion in 2.6., Select the Preferred Alternative, but is explicit when a compensatory technique is used, as described in 7.6., Multiple-Attribute Decision-Making.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	Tockey 2005 [3*]	Sommerville 2016 [10*]	Fairley 2009 [22*]
1. Software Engineering Economics Fundamentals			
1.1. Proposals	c3pp23-24		
1.2. Cash Flow	c3pp24-32		
1.3. Time-Value of Money	c5-6		
1.4. Equivalence	c7		
1.5. Bases for Comparison	c8		
1.6. Alternatives	c9		
1.7. Intangible Assets			
1.8. Business Model			
2. The Engineering Decision-Making Process			
2.1. Process Overview	c4pp35-36		
2.2. Understand the Real Problem	c4pp37-39		
2.3. Identify All Reasonable Technically Feasible Solutions	c4pp40-41		
2.4. Define the Selection Criteria	c4pp39-40, c26pp441-442		
2.5. Evaluate Each Alternative Against the Selection Criteria	c4pp41-42		

	Tockey 2005 [3*]	Sommerville 2016 [10*]	Fairley 2009 [22*]
2.6. Select the Preferred Alternative	c4p42, c26pp447- 458		
2.7. Monitor the Performance of the Selected Alternative	c4pp42-43		
3. For-Profit Decision-Making			
3.1. Minimum Acceptable Rate of Return	c10pp141- 143		
3.2. Economic Life	c11pp160- 164		
3.3. Planning Horizon	c11		
3.4. Replacement Decisions	c12pp171-178	c9	
3.5. Retirement Decisions	c12pp178- 181	c9	
3.6. Advanced For-Profit Decision Considerations	c13-17		
4. Nonprofit Decision-Making			
4.1. Benefit-Cost Analysis	c18pp303- 311		
4.2. Cost-Effectiveness Analysis	c18pp311- 314		
5. Present Economy Decision-Making			
5.1. Break-Even Analysis	c19		
5.2. Optimization Analysis	c20		
6. Multiple-Attribute Decision-Making			

	Tockey 2005 [3*]	Sommerville 2016 [10*]	Fairley 2009 [22*]
6.1. Compensatory Techniques	c26pp449-458		
6.2. Non-Compensatory Techniques	c26pp447-449		
7. Identifying and Characterizing Intangible Assets			
7.1. Identify Processes and Define Business Goals			
7.2. Identify Intangible Assets Linked with Business Goals			
7.3. Identify Software Products That Support Intangible Assets			
7.4. Define and Measure Indicators			
7.5. Intangible Asset Characterization			
7.6. Link Specific Intangible Assets with the Business Model			
7.7. Decision-Making			
8. Estimation			
8.1. Expert Judgment	c22pp367-369		
8.2. Analogy	c22pp369-371		
8.3. Decomposition	c22pp371-374		
8.4. Parametric	c22pp374-377		
8.5. Multiple Estimates	c22pp377-379		
9. Practical Considerations			

	Tockey 2005 [3*]	Sommerville 2016 [10*]	Fairley 2009 [22*]
9.1. Business Case			
9.2. Multiple-Currency Analysis			
9.3. Systems Thinking			
10. Related Concepts			
10.1. Accounting	c15pp234-245		
10.2. Cost and Costing	c15pp245-259		
10.3. Finance			
10.4. Controlling			
10.5. Efficiency and Effectiveness		c22pp422-23	
10.6. Productivity		c23pp689	
10.7. Product or Service			
10.8. Project			c2s2.4
10.9. Program			
10.10. Portfolio			
10.11. Product Life Cycle			
10.12. Project Life Cycle			
10.13. Price and Pricing		c23s23.1	
10.14. Prioritization			

[27].

This book provides an overview of quantitative methods in software engineering, starting with measurement theory and proceeding to performance management and business decision-making.

FURTHER READINGS

Project Management Institute,
A Guide to the Project Management Body of Knowledge (PMBOK® Guide) [24].

The *PMBOK® Guide* provides guidelines for managing individual projects and defines project management-related concepts. It also describes the project management life cycle and its related processes, as well as the project life cycle. It is a globally recognized guide for the project management profession.

Project Management Institute and IEEE Computer Society,
Software Extension to the Guide to the Project Management Body of Knowledge (SWX) [25].

SWX provides adaptations and extensions to the generic practices of project management documented in the *PMBOK® Guide* for managing software projects. The primary contribution of this extension to the *PMBOK® Guide* is its description of processes that are applicable to managing adaptive life cycle software projects.

B.W. Boehm, *Software Engineering Economics* [26].

This book is classic reading on software engineering economics. It provides an overview of business thinking in software engineering. Although the examples and figures are dated, it is still worth reading.

C. Ebert and R. Dumke, *Software Measurement*

D.J. Reifer,
Making the Software Business Case: Improvement by the Numbers [28].

This book is classic reading on making a business case in software and IT industries. Many useful examples illustrate how the business case is formulated and quantified.

REFERENCES

- [1] E. DeGarmo et al., *Engineering Economy*, 9th ed., Englewood Cliffs, NJ: Prentice Hall, 1993.
- [2] P. Rodriguez, C. Urquhart, and E. Mendes. “A Theory of Value for Value-based Feature Selection in Software Engineering,” *IEEE Transactions on Software Engineering*, 1, 2020.
- [3*] S. Tockey, *Return on Software: Maximizing the Return on Your Software Investment*, Boston, MA: Addison-Wesley, 2005.
- [4] *International Valuation Standards (IVS)*, 2019, Norwich: Page Bros, ISBN: 978-0-9931513-3-3-0.
- [5] K. Voigt, O. Buliga, and K. Michl, Business model pioneers: How innovators successfully implement new business models, 2017.
- [6] M.-I. Sanchez-Segura, G.-L. Dugarte-Peña, A. Amescua-Seco, and F. Medina-Domínguez, “Exploring how the intangible side of an organization impacts its business

- model,” *Kybernetes*, Vol. 50 No. 10, pp. 2790-2822. 2021.
<https://doi.org/10.1108/K-05-2020-0302>
- [7] ISO/IEC/IEEE International Standard – Systems and software engineering – Software life cycle processes – Part 2: Relation and mapping between ISO/IEC/IEEE 12207:2017 and ISO/IEC 12207:2008. IEEE, 2020, pp. 1-278, doi: 10.1109/IEEEESTD.2020.9238529.
- [8*] ISO/IEC/IEEE 15288 First edition 2015-05-15: ISO/IEC/IEEE International Standard – Systems and software engineering – System life cycle processes, IEEE.
- [9] T. Brown and B. Katz, *Change by Design: How Design Thinking Transforms Organizations and Inspires Innovation*, Revised and updated ed., New York, NY: Harper Collins, 2019.
- [10*] I. Sommerville, *Software Engineering*, 10th ed., New York: Addison-Wesley, 2016.
- [11] T. Gilb, *Competitive Engineering: A Handbook for Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage*, Oxford, UK: Elsevier Butterworth-Heinemann, 2005.
- [12] R. Kazman, M. Klein, and P. Clements, “ATAMSM: Method for Architecture Evaluation,” CMU/SEI-2000-TR-004, Software Engineering Institute, August 2000.
- [13] M.I. Sanchez-Segura, A. Ruiz-Robles, F. Medina-Domínguez, and G.L. Dugarte-Peña. “Strategic characterization of process assets based on asset quality and business impact,” *Industrial Management and Data Systems*, 117(8), 1720-1734.
- <https://doi.org/10.1108/IMDS-10-2016-0422>, 2017.
- [14] M.I. Sanchez-Segura, A. Ruiz Robles, F. Medina-Domínguez. “Uncovering hidden process assets: A case study.” *Information Systems Frontiers*. <https://www.springerprofessional.de/en/uncovering-hidden-process-assets-a-case-study/11724394>, 2016.
- [15] A. Osterwalder, Y. Pigneur, G. Bernarda, A. Smith, and T. Papadakos, *Value Proposition design*, Wiley, 2015.
- [16] D. Gotterbarn, K. Miller, and S. Rogerson, “Software engineering code of ethics,” *Commun. ACM* 40, 11, 110-118, doi: 10.1145/265684.265699, 1997.
- [17] S. McConnell, *Software Estimation Demystifying the Black Art*, 1st ed., Microsoft Press, 2009.
- [18] R.D. Stutzke, *Estimating software-intensive systems projects, products, and processes*, 1st ed., Addison-Wesley, 2005.
- [19] M. Ben-Eli, *Understanding Systems. Systems Innovation*, <http://bit.ly/2XNlh3D>, 2019.
- [20] J. Sterman, *Business Dynamics: Systems Thinking and Modeling for a Complex World*, McGraw-Hill, 2000.
- [21] S. Pereira, G. Medina, et al., “System thinking and business model canvas for collaborative business models design,” *IFIP Advances in Information and Communication Technology*, Vol. 488, pp. 461-468, 2016.
- [22*] R.E. Fairley, *Managing and Leading Software Projects*, Wiley-IEEE Computer Society Press, 2009.

- [23] Project Management Institute, Inc.,
PMI Lexicon of Project Management Term,
2012.
- [24] Project Management Institute,
A Guide to the Project Management Body of Knowledge (PMBOK® Guide), 5th ed.,
Project Management Institute, 2013.
- [25] Project Management Institute and IEEE
Computer Society,
Software Extension to the PMBOK® Guide Fifth Edition, ed: Project Management
Institute, 2013.
- [26] B.W. Boehm,
Software Engineering Economics, Prentice-Hall, 1981.
- [27] C. Ebert and R. Dumke,
Software Measurement, Springer, 2007.
- [28] D.J. Reifer,
Making the Software Business Case: Improvement by the Numbers, Addison-Wesley,
2002.

CHAPTER 16

COMPUTING FOUNDATIONS

ACRONYMS

Acronym	Definitions
ADT	Abstract Data Type
AI	Artificial Intelligence
ANSI	American National Standards Institute
AVL Tree	Adelson-Velskii and Landis Tree
BCNF	Boyce-Codd Normal Form
BST	Binary Search Tree
CASE	Common Application Service Element
CDRAM	Cache DRAM
CERT	Computer Engineering Response Team
CISC	Complex Instruction Set Computer
CRUD	Create, Read, Update, Delete
CUDA	Compute Unified Device Architecture
DAG	Direct Acrylic Graph
DAL	Database Access Language
DAS	Direct Access Storage
DBCS	Double Byte Character Set
DCL	Data Control Language
DDL	Data Definition Language
DDR SDRAM	Double data rate SDRAM
DKNF	Domain/Key Normal Form
DMA	Direct Memory Access
DML	Data Manipulation Language
EDW	Enterprise Data Warehouse
FCFS	first come, first served

FIFO	First In, First Out
FPU	Floating Point Unit
HCI	Human-Computer Interface
HMPP	Hybrid Multicore Parallel Programming
HTTP	Hyper Text Transfer Protocol
IPC	Inter-Process Communication
ISA	Instruction Set Architecture
MIMD	Multiple instruction, multiple data stream
MISD	Multiple instruction, single data stream
MISRA	Motor Industry Software Reliability Association
ML	Machine Learning
NAS	Network Access Storage
OSI	Open Systems Interconnection
PDU	Protocol Data Unit
RDBMS	Relational DBMS
RDM	Runtime Database Manager
RDRAM	Rambus DRAM
RISC	Reduced Instruction Set Computer
RTOS	Real Time Operating System
SAN	Storage Area Network
SASE	Specific Application Service Element
SDRAM	Synchronous DRAM
SEI	Software Engineering Institute
SIMD	Single instruction, multiple data stream
SISD	Single instruction, single data stream
SQL	Structured Query Language
SRTF	Shortest Remaining Time First

Introduction

Software engineers must understand and internalize the differences between their role

and that of a computer programmer. A typical programmer converts a given algorithm into a set of computer instructions, compiles the code, creates links with relevant libraries, binds, loads the program into the desired system, executes the program, and generates output.

On the other hand, a software engineer studies the requirements, architects and designs major system blocks, and identifies optimal algorithms, communication mechanisms, performance criteria, test and acceptance plans, maintenance methodologies, engineering processes and methods appropriate to the applications and so on.

The key purpose of the *Software Engineering Body of Knowledge (SWEBOk) Guide* is to

identify the areas of knowledge that professional software engineers must know, according to practicing subject matter experts worldwide.

Software engineers are expected to have deep and broad knowledge of various concepts of computer science and be able to apply them. These concepts form the foundations of computing.

BREAKDOWN OF TOPICS FOR COMPUTING FOUNDATIONS

The breakdown of topics for the Computing Foundations Knowledge Area (KA) is shown in Figure 16.1.

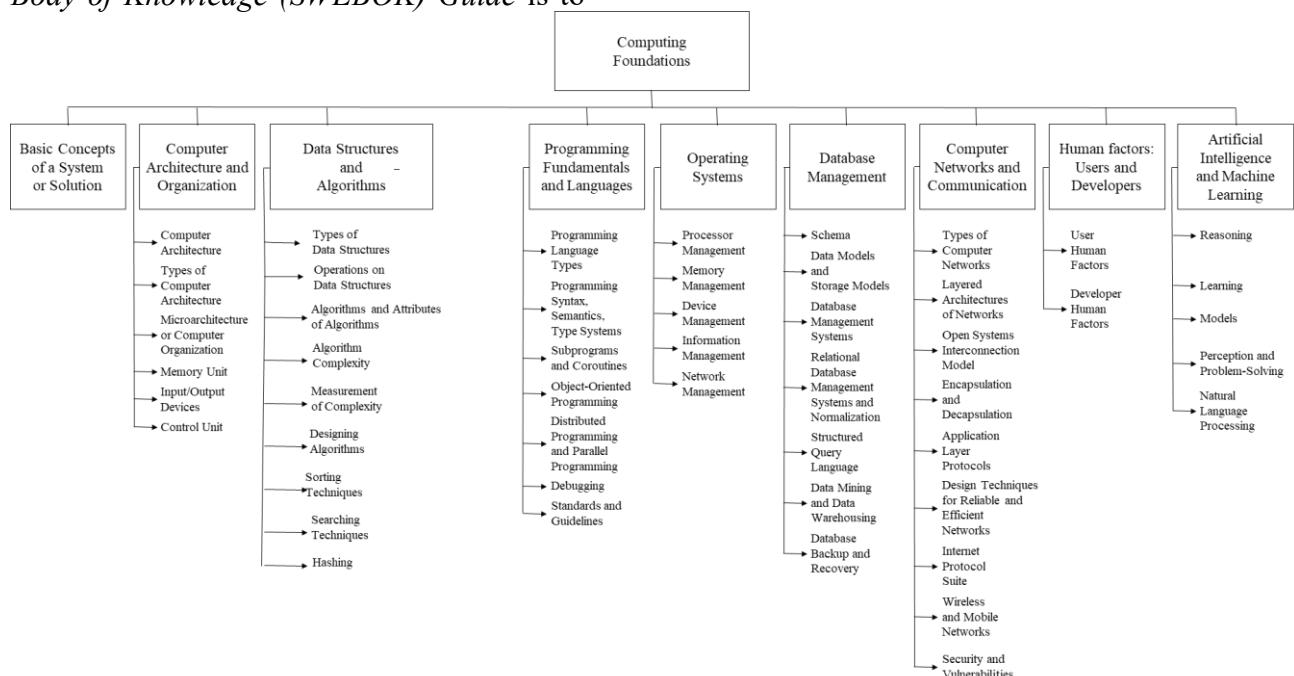


Figure 16.1. Breakdown of Topics for the Computing Foundations KA

1 Basic Concepts of a System or Solution

The problem to be solved has to be analyzed in greater detail for functional requirements, user interactions, performance requirements, device interfaces, security, vulnerability, durability and upgradability. A system is an integrated set of subsystems, modules and components that perform specific functions independently. Delineating the problem and solution is critical.

An engineered system ensures the subsystems are designed to be:

- **Modular:** Each subsystem (module) is uniform (similar size).
- **Cohesive:** Each subsystem performs one specific task. Ideally, systems should be highly cohesive.
- **Coupled:** Each subsystem functions independently, as much as possible. Ideally, systems should be loosely coupled.

The subsystems may further be broken down into modules and sub-modules that also exhibit these characteristics.

The system may include both software and hardware subsystems. The hardware must be designed to support the software subsystems and satisfy all user requirements, especially user interfaces (input/output (I/O)) and performance.

This section focuses on designing and building engineered software subsystems.

The applications may require systems that are manual or fully or semiautomated; real-time, online or offline; distributed or single-location, and so on.

The software subsystems' architects have to consider appropriate technology, tools, data structure, operating system, database (if required), user interfaces, programming languages, and algorithms for computing solutions optimally among others.

Software requirements, architecture, design, construction, testing, methods and models, quality assurance, and security are discussed in detail in other chapters as independent KAs.

The Computing Foundations KA focuses on explaining the key computer science concepts a software engineer has to know well to architect, design, construct, deploy and maintain useful, high-quality software subsystems.

2 Computer Architecture and Organization

Computer architecture refers to the components of a computer system designed for specific purposes. Computer organization explains how the units within the system connect and interact to achieve those purposes.

System architects must analyze the application for which the computer system is to be designed or developed; identify the critical components, including I/O devices required (along with throughput), types and quantum of memory, processing power, and coprocessors required; and choose or design appropriate computer architecture and organization. Contingencies should be built in for the resources required.

This content area discusses various computer architectures and organizations a system or software architect needs to know.

2.1 Computer Architecture

Architecture describes what the computer or system does, and its components, such as memory, data storage devices, graphics, and the computers or processor's computing power. A computing system typically has memory, I/O devices and a central processing unit (CPU).

These components are connected through physical signal lines called a *bus*. Typically, three types of buses are used for specific purposes:

- Address bus, which addresses or accesses a specific memory location or I/O device

- Data bus, which stores (writes) or retrieves (reads) data to and from the memory location
- Control bus, which provides control signals from the CPU to I/O devices (read or write, enable or disable, interrupt, status, reset, etc.)

Software engineers are expected to know the details of the functioning and timing of different types of buses — first-generation, second-generation and third-generation buses; internal and external buses; serial and parallel buses; simplex, full-duplex and half-duplex buses; Mil-Std-1553Bbus, Wishbone buses, etc.

2.2 Types of Computer Architectures

2.2.1 Von Neumann Architecture

John von Neumann designed a computer system architecture with five essential components as shown in Figure 16.2:

- Arithmetic Logic Unit (ALU) that performs arithmetic and logic computation.
- Memory where the program and data are loaded and executed (program and data reside in the same memory space).
- Input devices (e.g., keyboard, mouse, serial port, hard disk) that allow the user to provide inputs and control commands.
- Output devices (e.g., monitor, printer) that transmit or communicate the computed results.
- The control unit synchronizes all devices, memory and ALU.

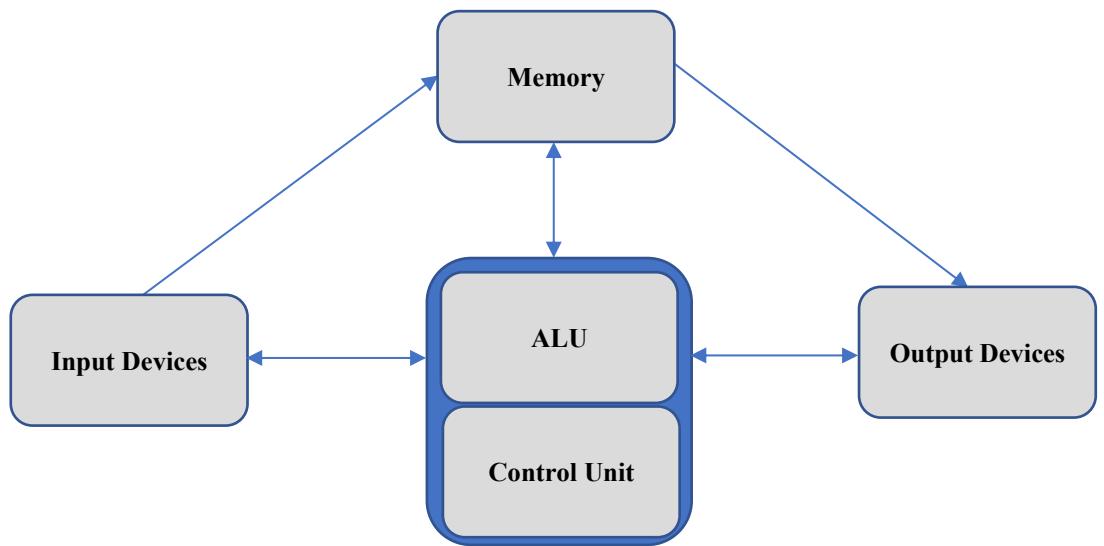


Figure 16.2. Computer Architecture

2.2.2 Harvard Architecture

The Harvard architecture provides separate memory blocks for code (program or instructions) and data. As the code and data memory blocks are different, the contents of address 0000 in the code block and the contents of address 0000 in the data block are different. The CPU reads instructions from the code addresses and reads data from the data addresses.

The system design and implementation in the original Harvard architecture were relatively complex. The modified Harvard architecture provides one memory block but partitions it into code and data sections. Data memory sections are read/write capable, and code memory sections are read-only (thus protects code from getting corrupted at runtime). I/O operations can be performed simultaneously.

2.2.3 Instruction Set Architecture

An instruction set architecture (ISA) is an abstract model of how a CPU executes the instruction sets defined for

the system. An ISA defines registers (address, data, flags), data types, instructions specific to the computer or system, memory (internal and external) addressing schemes, and I/O handling models.

A Reduced Instruction Set Computer (RISC) architecture and a complex instruction set computer (CISC) architecture are the two primary types of ISAs.

In RISC, the instructions perform single tasks such as reading from memory or I/O, performing arithmetic or logical computation, and storing data into memory or I/O. The computer system is simple but requires more instructions to execute a task. It requires fewer clock cycles per instruction, and instruction sizes tend to be fixed. As the instruction set is small (fewer instructions), it is easier to build a compiler, and the program can be relatively large. RISC architectures are typically designed for general-purpose processors.

The instructions are relatively more powerful in CISC and can perform multiple tasks such as reading data from memory + performing arithmetic operation + storing the result in memory. Here, fewer instructions are required to perform a task, but the instructions take more clock cycles to complete. Instruction sizes vary widely depending on operations with registers, memory and I/O. Programs are relatively small. CISCs are typically designed for specific purposes such as digital signal processing (DSP) and graphics.

2.2.4 Flynn's Architecture or Taxonomy

The computing architectures described above consider a single computer at a time. Michael J. Flynn proposed concurrent computer architectures, where multiple instruction streams and multiple data streams are used in the system. Software engineers need to know the different types of Flynn's architecture, with examples, including the following:

- Single instruction, single data stream (SISD) architecture
- Single instruction, multiple data stream (SIMD) architecture
- Multiple instruction, single data stream (MISD) architecture
- Multiple instruction, multiple data stream (MIMD) architecture

Variants of these architectures include array processing, parallel processing, and associate processing; processing single program multiple data streams, and multiple program multiple data streams. Software engineers are expected to know the differences

among these architectures, along with case studies, so that they can choose the right architecture to solve the problem at hand.

2.2.5 System Architecture

System architecture is the overall system design, considering hardware architecture, software architecture, modules, interfaces, data management, and communication among modules. Distributed computing has become affordable with the development of efficient, high-end, high-performance servers, storage, network devices, software, and tools. Several reference designs or architectures are available for any given application.

Typical system architectures include the following:

- **Integrated system architecture:** Computing, I/O, data and networking are tightly coupled and available in one box. This architecture is typically used in solutions designed for specific applications.
- **Distributed system architecture:** Computing and storage are located in separate but networked boxes. This architecture supports scaling, provides centralized or isolated data storage, and shares computation load.
- **Pooled system architecture:** Several computing, storage and network resources are available in pools and provided depending on demand. This architecture provides for efficient use of shared resources.

- **Converged system architecture:** As the name implies, this is the convergence of distributed and pooled architectures. This architecture supports agility and scalability.

Software engineers are also expected to know and be able to apply various other architectures, including .NET Framework architecture, Unix architecture, and virtual machine architecture.

2.3 Microarchitecture or Computer Organization

Microarchitecture or computer organization explains how the ISA of a computer is implemented and how different components in the system function and interact with one another to produce the desired outcome.

System architects and engineers must know the various components used in the system along with how they function. Some of these components are discussed below.

2.3.1 Arithmetic Logic Unit

The ALU performs all arithmetic computations and logical operations. The CPU typically has an ALU, processor, memory, and control unit. High-end CPUs may also have other functionality-specific processing units, such as a floating-point unit (FPU), to perform computations involving floating point or real numbers (fractions). ALUs have registers that are high-speed memory and internal to the ALU. The ALU executes the processor instruction sets. All operations are typically carried out on the registers.

Various schemes may be implemented to improve the performance of the ALU, including pipeline processing and parallel processing. The latest CPUs provide multiple cores and multiple threads that help achieve maximum throughput. Software engineers are expected to know the differences between multiple cores and multiple threads, along with specific cases illustrating the best use of these.

Specific-purpose coprocessors and associate processors are used with main processors to support faster processing.

2.3.2 Memory Unit

Memory units are used to store data or information, which is accessed by the CPU. The total amount of memory a computer can have is derived from the maximum number of address lines supported by the CPU. Different types of memory used in the system include read-only memory (ROM), and read-write memory or random access memory (RAM). ROM is also RAM.

Software engineers working on performance-critical applications are expected to know the differences among various types of memory, including static RAM (SRAM), dynamic RAM (DRAM), asynchronous DRAM (ADRAM), synchronous DRAM (SDRAM), double-data-rate SDRAM (DDR SDRAM), rambus DRAM (RDRAM), and cache DRAM (CDRAM), along with pros, cons and use cases of each.

2.3.3 Input/Output Devices

As the names imply, input devices are those that provide inputs to the computer system, and output devices are those that deliver computer systems' output to the user. While

some devices are input only (keyboard, mouse, microphone, etc.) or output only (printer, monitor, speakers, etc.), a few devices serve as both input and output devices (e.g., touch screens, hard disks, USB drives).

Software engineers are expected to understand the interface of the I/O devices with the system, whether they are memory-mapped I/O or I/O-mapped I/O devices, and device drivers required for the users or applications to interact with the devices through the operating system.

2.3.4 Control Unit

The control unit synchronizes multiple components in the computer system. Typically, control units are part of the CPU. They interpret instructions and coordinate data movement among different components (memory, I/O devices and ALU). Control units are also used to enable or disable components or devices and reset devices.

Software engineers are expected to be aware of the different types of control units, including hardware control units and micro programmable control units (single-level and two-level control stores), along with the benefits and challenges of each.

3 Data Structures and Algorithms

Data structures are fundamental to computer science and software engineering. Every program uses data — receives input (data), performs specific functions on the data and produces output. Data structures is about representing different types of data effectively, performing various operations on the data proficiently, and storing and retrieving data efficiently. Software engineers must internalize data structures, the selection of data structures, and operations on them specific to applications.

In this chapter, different types of data structures and various operations on them are discussed.

3.1 Types of Data Structures

Data type is an attribute of data. Various data types are identified and defined based on different characteristics of data, the need for grouping data items and various operations performed on data. Data structures are grouped primarily based on the physical and logical ordering of data items.

Primarily, data is grouped into three types: basic, composite or compound, and abstract.

Basic or primitive data types include character, integer, float or real, Boolean, and pointer data.

Compound data types are made of multiple basic or primitive, or even multiple compound data types. Some of the compound data types include sets, graphs, records and partitions.

An Abstract Data Type (ADT) is defined by its behavior (semantics) from the user's perspective, specifically from the point of possible values and operations.

Composite or compound data types are further grouped under linear, and hierarchical or nonlinear data types.

Linear data types include one-dimensional and multidimensional arrays, strings, linked lists (singly linked lists, doubly linked lists, circular lists), stacks, queues, and hash tables.

Hierarchical or nonlinear data types include trees, binary trees, n-array trees, B trees, B+ trees, weighted balanced trees, red-black trees, graphs, heaps, binary heaps and graphs.

In the current era of free text queries or natural language processing, software engineers may need to understand strings and various operations on strings, and to be able to analyze skip lists.

Software engineers must understand the nuances of various types of data and their sizes in memory (short integer, integer, long integer, long long integer, signed and unsigned integer, float, double, long double, double byte character set (DBCS), Boolean, etc.), along with how various data types are represented and stored in memory and how various operations are performed on them. Sets, graphs, and trees are discussed in more detail in the Mathematical Foundations KA.

3.2 Operations on Data Structures

Basic operations performed on data structures include Create, Read, Update and Delete (CRUD). Compound data types also require various ways of traversing data sets to

identify specific data items before performing the operation.

It is important to ensure that any insertion or deletion of items in a data set or database does not alter the data set or database in a way that violates any policy under which the database was designed and built.

Additional operations performed on data structures include sorting the data items in a specific order, searching and locating a data item, and merging two or more data sets into one set without disturbing the policy on which the data set is built. Searching and sorting algorithms are discussed in the next section.

Different data structures are created to suit specific applications, such as stacks, queues, trees, and graphs. Software engineers are encouraged to learn the traversals through nonlinear data structures, which include different tree parsers (pre-order, in-order, and post-order tree traversals), CRUD operations on trees, tree balancing, Binary Search Trees (BSTs), AVL trees, and red-black trees, and to learn tree search algorithms (depth first, breadth first, shortest paths, etc.). Some of these are discussed in the Mathematical Foundations KA.

3.3 Algorithms and Attributes of Algorithms

All software implements logic to perform the required function. That logic or algorithm to perform a specific task has to be designed or chosen with consideration for system performance, security, portability, maintainability, scalability and simplicity, among other concerns.

The complexity of an algorithm is determined by measuring the

computational resources (computing power and space) consumed by that algorithm for a given set of data.

A thorough understanding of data structures is vital for analyzing and designing good algorithms. Refer to the “Data Structures and Organization” content area for more details.

The attributes of algorithms are many and include functionality, correctness, robustness, modularity, maintainability, programmer-friendliness (ease of integration into the project and ease of use), user-friendliness (i.e., how easily it is understood by people), need for programmer time, simplicity, and extensibility.

A commonly emphasized attribute of algorithms is “performance” or “efficiency.”

The parameters that matter for an algorithm’s resource consumption include, but are not limited to:

1. Hardware
2. Software
3. Algorithm selection and design for a specific problem
4. Effective implementation

3.4 Algorithm Complexity

The complexity of an algorithm is a measure of the resources it consumes (computing power or memory) for a specific problem and given data set.

Choosing the right data structures and operations on data structures and ensuring optimal implementation of the algorithm also effect the algorithm’s complexity.

3.5 Measurement of Complexity

Often, the complexity of an algorithm is denoted by the resources consumed in the worst-case scenario. The complexity of algorithms is typically measured by asymptotic notations for

best-case, worst-case and average-case scenarios in terms of resource consumption for a given data set.

Popular asymptotic notations for algorithms are listed in Table 16.1.

Asymptotic Notations	Description
Big O	Big O notation provides the upper bound of operations (worst-case scenario) for a function $f(n)$.
little-o	Little o notations are used to depict scenarios where the upper bound is not tight.
Big Omega (Ω)	Big Ω notations are used to depict lower bounds (best-case scenarios) for a function $f(n)$.
little-omega (ω)	Little omega (ω) notations are used to depict loosely bound best-case scenarios of an algorithm.
Theta (Θ)	Theta notation bounds the function from above and below (provides average-case complexity of an algorithm).

Table 16.1. Asymptotic Notations of Algorithms

Learning the computation of the listed notations for different sets of input data (e.g., sorted, unsorted, and sorted in reverse order) is important.

The complexity of an algorithm can be constant, linear, quadratic, cubic,

exponential or logarithmic. These complexities are described in Table 16.2. Typically, constants are not considered when computing the efficiency of an algorithm.

Complexity	Notation	Description
Constant	$O(1)$	Regardless of the data size, the algorithm takes a constant number of steps to perform the operation.
Linear	$O(n)$	The number of operations is linearly proportional (steps are a constant multiple of the data set size n).
Quadratic	$O(n^2)$	The algorithm takes the order of n^2 steps for performing the operation on the data set of size n .
Cubic	$O(n^3)$	The algorithm takes the order of n^3 steps for performing the operation on a data set size of n .
Exponential	$O(n^k)$ $O(2^n)$ $O(n!)$	The algorithm has an order of exponential dependability for performing the operation on a data set of size n .

Logarithmic	$O(\log(n))$ $O(N \cdot \log(n))$	The algorithm takes the order of $\log(n)$ steps (base of log is typically 2).
-------------	--------------------------------------	--

Table 16.2. List of Algorithmic Complexities

3.6 Designing Algorithms

The software engineer must consider the specific application's purpose and the performance requirements in order to select an appropriate algorithm. In addition, the software engineer must consider linear programming versus parallel programming and single-versus multi-threading.

The efficiency of an algorithm is measured by the resources it consumes, primarily computing time and memory.

A software engineer has to know a few standard algorithms and relevant concepts, including the following:

- Common types of algorithms:
Brute force algorithm, Recursive algorithm, Divide & Conquer algorithm, Dynamic programming algorithms, Greedy algorithm, Backtracking algorithms, Randomized algorithms.
- Randomized approximation algorithms, randomized rounding, approximation algorithms, P and NP complexity class algorithms, Cook's theorem, reductions and completeness algorithms.
- Multiple comparison operations performed simultaneously in a network model of computation. Popular sorting network algorithms include comparison networks, zero-one principle, merging network and bitonic sorter.

- Optimized algorithms for performing several operations on a matrix, such as matrix multiplication, transposition, matrix inversion, median, and finding determinants.
- Cryptographic complexity and algorithms: secret key (symmetric) encryption algorithms, public key (asymmetric) encryption algorithms and hash functions.
- One-way functions, class UP, space complexity, deterministic and nondeterministic space complexity classes, the reachability method, and Savitch's theorem.
- Graph representations, graph algorithms, breadth-first and depth-first search, topological sort, minimum spanning tree, Kruskal and Prim algorithms, and single-source shortest paths (Bellman-Ford and Dijkstra algorithms).
- Complexity of randomized computation, interactive proofs, complexity of counting, Boolean circuit complexity.

Of particular importance in many software systems are algorithms for sorting and searching, these are discussed in more detail.

3.7 Sorting Techniques

Sorting is the process of arranging data items in a specific order.

Popular sorting algorithms include Linear sort, Bubble sort, Quick sort, Merge sort, Radix sort, Heap sort, Bucket sort, Pigeonhole sort, Bitonic sort, Tree sort, Cartesian Tree sort, 3-Way Quick sort, 3-Way Merge sort, and Sorting Singly / Doubly linked lists.

Each sorting algorithm has its benefits and shortfalls. Selection of an appropriate algorithm depends on the size of input data, the type of data (linear or nonlinear), and the type of data set (completely unsorted, partially sorted, etc.). The algorithms are implemented in both iterative and recursive methods. Typically, iterative methods are better than recursive methods for CPU performance and memory. However, recursion provides easy methods for solving specific problems, such as tree operations. If adequate computing power and memory are available, the difference between recursive and iterative implementation methods is negligible.

In the case of applications where certain sorting algorithms work best, software engineers should learn and accommodate any preconditions and complexities (demand on memory and computing power) involved in using them.

3.8 Searching Techniques

Searching is a process of finding specific data items or records in a set of data items or a database.

Search algorithms are primarily categorized into sequential search (data set is traversed sequentially until the end of the data set) and interval search (the search moves efficiently through a sorted list, balanced tree, etc.), based on how data sets are organized.

Depending on the type of the data item and the size of the data set, various search techniques are used to find the desired data item. Popular search algorithms include linear, binary, jump, interpolation, exponential, Fibonacci, sub-list (search a linked list in another list), logarithmic, tree and hashing.

3.9 Hashing

Hashing is one of the very important and popular technique in which data of arbitrary size (key values) are converted into values of fixed size called *hash values*, which index into a hash table so the data records can be located easily. The function used for that purpose is called a hash function, and the values returned are called hash values, hash codes, digests, or hash keys.

Different properties of hash functions, such as uniformity, efficiency, universality, applicability, deterministic, defined or variable range, data normalization, testing, and measurement, must be understood and considered when designing or choosing a hash function.

Various types of hash functions are designed for different types of key values, applications, and database sizes. Hash function types include trivial hash function, division method, mid-square method, digit folding method, multiplicative hashing, double hashing, open and closed hashing, rehashing, extendible hashing, and cryptographic and noncryptographic hash functions.

Software engineers are expected to learn, implement and be able to compare different types of hashing algorithms, various collision resolution techniques, linear probing, quadratic

probing, separate chaining, and open addressing.

4 Programming Fundamentals and Languages

Computer programs are sequential steps or instructions that work on provided inputs and generate desired or specific outputs.

Software engineers must carefully consider various aspects before selecting a programming language to solve a specific problem.

4.1 Programming Language Types

Depending on the hardware, operating system, and application various types of programming languages are developed and used. Basic types of programming languages include microprogramming, machine languages, assembly programming and high-level programming.

Microprogramming is executed within the microcontroller or microprocessor chips to execute the assembly language instructions.

Assembly language programs use the mnemonic specified by the microcontroller or microprocessor. Typically, the microcontrollers or microprocessors are designed to address specific applications (DSP processors, graphics chips, I/O controllers, mathematical coprocessors, generic processors, etc.).

High-level languages enable programs to be written in instructions similar to English, which makes it easy for the developer and maintainer to write and maintain the programs. Various high-level programming languages include the following:

- Functional programming languages
- Procedural programming languages
- Object-oriented programming languages
- Scripting languages
- Logic programming languages

Software engineers need to study multiple programming languages to choose the right one for a specific application.

Many programming languages, such as C, C++ and Java, use compilers to build executables, whereas other programming languages, such as JavaScript, Ruby and Python, use interpreters.

4.2 Programming Syntax, Semantics, Type Systems

The syntax of a programming language is its grammar — the various constructs the programming language uses. A compiler or interpreter checks the syntax of all declarations, statements (algorithmic statements, conditional or logical statements, control statements, loops, special language-specific statements, macros, etc.), and functions or procedures, and creates notifications of any errors.

Semantics refers to the meaning or interpretation of the statement. The meaning could vary at runtime, depending on runtime values.

A type system assigns a type to a data item or to constructs of a program, such as variables, expressions and functions. In static typing, the type is fixed; it is defined during program creation and checked at compilation time. Languages such as C, C++ and Java support static typing. In dynamic

typing, the type of a variable can change at runtime depending on the context and hence is checked at runtime. Dynamic typing languages include Python, Perl, PHP and Ruby. Dynamic typing is also called polymorphic typing.

Software engineers are expected to know how high-level programming languages are translated into machine languages, to be familiar with the various types of compilers, and to know the differences among compilers, interpreters, cross-compilers, assemblers and cross-assemblers. Software engineers are encouraged to learn about compiler phases, including preprocessing, lexical analysis, syntax analysis, intermediate code generation, optimization, code generator, linkers, loaders and debuggers.

Tokens, grammars, syntax trees, parse trees and weights to various operators (precedence) in arithmetic and logical equations are important to analyze and understand.

4.3 Subprograms and Coroutines

Subprograms or functions are programs or building blocks that perform specific (part) functions in the scope of a complete project. Subprograms provide for breaking the larger program into smaller modules. The modules are typically sections of code that are used multiple times in multiple places. The subprograms reduce memory space, improve readability and maintainability of the program, and execute parts of the program with different values at different places and times.

The subprograms have an entry point and typically have multiple input parameters on which the subprogram acts and produces output. The scope of

input parameters is local to the subprogram. Subprograms that return value by their name (which can be used as a variable in a statement) are called *functions*, and subprograms designed not to return any value are called *procedures*.

By default, the scope of subprogram parameters is dynamic and local to the subprogram. However, if the subprograms have to remember their history or previous values, they have to be declared static or as specified in the chosen programming language.

Different programming languages support one or more types of parameters' passing, including pass-by-value, pass-by-reference, pass-by-name, pass-by-result and pass-by-result-value. Software engineers should know the differences among these types and use them appropriately.

Many high-end languages support the nesting of subroutines and recursions, where a subroutine calls itself. Different types of recursions include cyclic or direct recursion (subroutine calls itself) and acyclic or indirect recursion (subroutine A calls subroutine B, which in turn calls subroutine A). It is important to establish the exit criteria in recursive subprograms.

Software engineers are encouraged to understand, using case studies, how the subprogram return address and parameters are stored in memory (runtime stack), how they are used in the subprogram and for returning to the called subprogram, and the scope of variables (global and local).

A subprogram with multiple entry points, where the previous exit point is remembered for resumption at a later point, is called a **Coroutine**. A

Coroutine call is typically called a *resume call*. The first resume call enters the subroutine from the beginning, and subsequent resume calls enter the subroutine at the point where it was exited last.

High-end languages that support Coroutines include C++20, C#, Java, JavaScript, Kotlin, Perl, .NET Framework, Python, Ruby and many assembly languages.

Software engineers are encouraged to understand specific applications where coroutines are useful and to use the coroutines. It is an interesting exercise to implement coroutines in C, as C does not support coroutines natively.

Figure 16.3 depicts the functioning or control flow of coroutines.

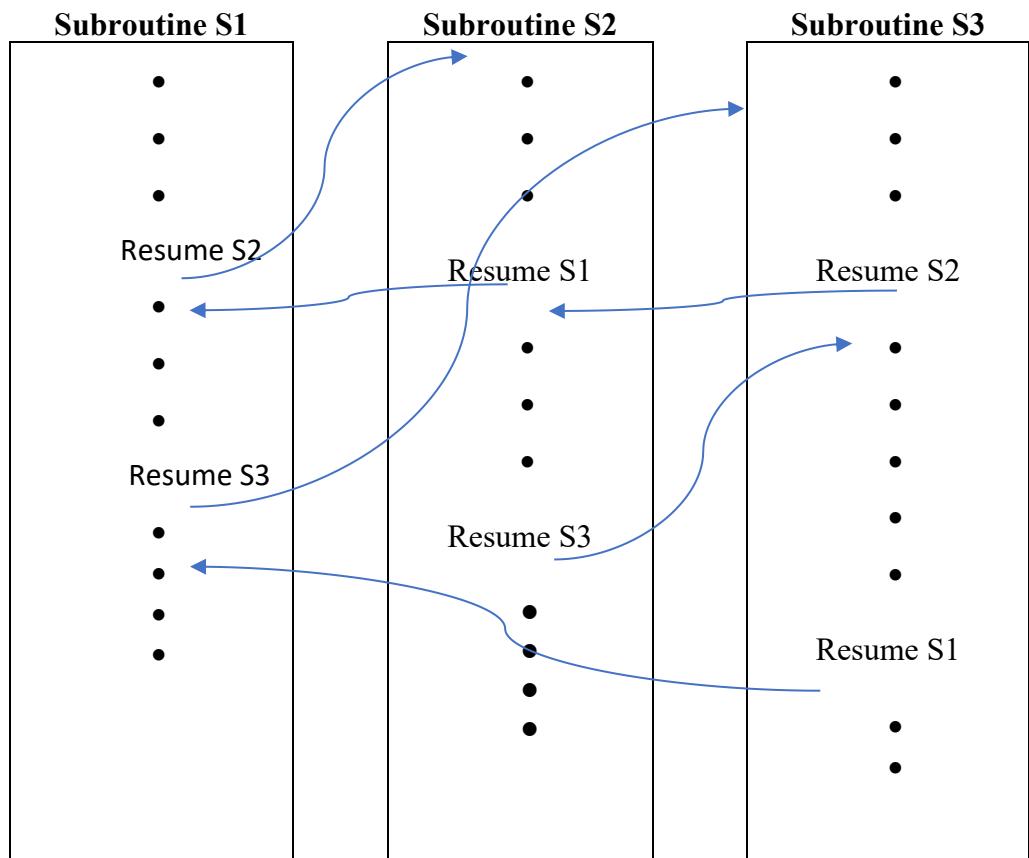


Figure 16.3. Example of Coroutine

4.4 Object-Oriented Programming

As the name suggests, object-oriented programming languages are based on objects. The objects typically have both data and functions that operate on that data. The data of an object is typically called the object's attributes or properties, and the code or functions that work on the attributes are called operations externally (by the client or

user) and called methods internally (referring to how the operation is implemented by the developer).

A Class is a programmer-defined prototype that defines the attributes and methods. Objects are actual instances of a Class. There could be multiple Objects of a Class with varied characteristics. For example, a Class can be defined by the characteristics

and operations of a vehicle, whereas objects are instances of the class vehicle such as car, bus or truck.

The objects interact with one another using the methods or operations.

Important characteristics of object-oriented programming (OOP) are Abstraction, Encapsulation, Inheritance and Polymorphism.

Abstraction is a property that exposes only required or relevant information to the user, hiding the details and nonessentials. Typically, a superclass is created with generalized methods, and the subclasses implement the methods as appropriate for specific instances of the class (object). Thus, the implementation is hidden from the user of the superclass.

One of the key benefits of encapsulation is the ability to hide or protect data from unauthorized users. The software engineer can give different levels of protection to data and methods by declaring them private (local to class) or public (available to other classes). This also protects data from corruption, either intentional or accidental.

Inheritance is an important feature of OOP, where a subclass or derived class inherits the properties of a superclass or base class. Primary inheritance modes include public, protected and private modes.

Polymorphism is another key feature of OOP. Polymorphism is the ability to have multiple forms depending on the object. For example, shape could be a base class with draw as a method, and objects could be a circle, triangle or rectangle. The implementation of method draw, though the name is the same, differs for a circle, triangle and

rectangle. Polymorphism has two types:

- **Static or compile-time polymorphism:** The methods (functions) or operators are overloaded and resolved during compile time. Example: The methods, though they have the same name, will have different types or numbers of parameters.
- **Dynamic or runtime polymorphism:** The overloaded method to be executed is resolved at runtime. Example: When both base class and derived class have the same method, the base class method is said to be overridden.

Popular OOP languages include C++, C#, Cobol 2002, Java, Python, Lisp, Perl, Object Pascal, Ruby and Smalltalk.

It's important to recognize that using OOP requires a different mindset than using traditional, procedural, or structured programming does.

4.5 Distributed Programming and Parallel Programming

In a distributed computer system, multiple parts of the software are run on multiple computers, connected through computer networks, to achieve a common goal. Writing such programs is called *distributed programming*.

Parallel programming is a type of computing in which different parts of the program are run in parallel to achieve the same objective or goal. Table 16.3 compares distributed and parallel programming.

Parameters	Distributed Programming	Parallel Programming
Functionality	A task is shared and executed by multiple computers that are networked.	One or more processors on a computer share and execute the task in parallel.
Computers	Multiple computers in different locations but networked.	One computer with one or more processors or cores.
Memory	Each computer has its own memory.	Computers can have shared or distributed memory.
Communication	Computers communicate through networks.	Processes communicate through a bus or inter-process communication (IPC) methods.
Benefits	<p>Failure of one computer does not affect the functioning of the task, as it is transferred to another computer.</p> <p>Provides scalability and reliability for end users.</p>	<p>As multiple processes run in parallel, CPU performance increases.</p> <p>Failure of one processor does not affect the performance of other processors or cores.</p>
Disadvantages	<p>Having multiple systems could become expensive; the cost must be weighed against customers' need for application uptime.</p> <p>Network delays could affect the overall functioning of the task.</p> <p>Designing an efficient distributed computing system is relatively difficult.</p>	<p>Using multiple processors or cores could be expensive.</p> <p>Dependency of one process on another process could introduce latency.</p>
Example Applications	Telephone and cellular networks, internet, World Wide Web networks, distributed database management systems, network file systems, grid computing, cloud computing.	Distributed rendering in computer graphics, scientific computing.
Example Programming Languages	Golang, Elixir, Scala.	Apache Hadoop, Apache Spark, Apache Flink, Apache Beam, CUDA, OpenCL, OpenHMPP, OpenMP for C, C++ and Fortran.

Table 16.3. Comparison of Distributed and Parallel Programming

4.6 Debugging

Programs, when written, are expected to function properly and generate the

expected output. However, programmers often face three types of errors — syntax errors, runtime errors, and logical errors — at different stages of software development.

Syntax errors are deviations from the standard format specified by programming languages. These are explicitly identified by compilers and are easy to fix.

Runtime errors surface when a program runs into an unexpected condition or situation such as dividing by zero, memory overflow, or addressing a wrong or unauthorized memory location or device, or when a program tries to perform an illegitimate or unauthorized operation or tries to access a library, for example. The programs must be thoroughly tested for various types of inputs (valid data sets, invalid data sets and boundary value data sets) and conditions to identify these errors. Once identified, runtime errors are easy to fix.

Logical errors are slipups in implementing the logic to achieve the desired output. These errors must be traced and resolved with various data for each functionality. Several sophisticated high-end debuggers help trace each variable or data item and support setting various types of break points.

4.7 Standards and Guidelines

As the computing system or application becomes bigger and complex, more programmers are involved. Their individual programming styles affect the project schedules and make system integration difficult, so systems become defect-prone, and maintenance and enhancement become challenging.

An estimated 82% of vulnerabilities are caused by clashes between programming styles (<https://www.ptsecurity.com/www-en/analytics/web-vulnerabilities-2020/>).

Hence, quality-conscious companies often have defined standards and guidelines, which set rules and recommendations for their programmers and testers to follow.

When software teams follow appropriate coding standards, they create readable, cleaner, portable, reusable, modular, easily maintainable, less defect-prone software code, and project schedules become more predictable. The following practices can help organizations implement such standards successfully:

- Carefully choose the coding standards and guidelines that suit the application or system being developed.
- Consider open standards created by community participation, such as Software Engineering Institute (SEI) Computer Emergency Response Team (CERT), as well as closed standards created by working groups such as the Motor Industry Software Reliability Association (MISRA).
- Educate programmers to follow adopted standards and guidelines.
- Use tools and periodic reviews to ensure adopted standards and guidelines are followed.
- Review and revise standards and guidelines from time to time, learning from project execution.

SC 22 is a subcommittee of the Joint Technical Committee ISO/IEC JTC 1

of the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) for defining standards for programming languages, their environments and system software interfaces (**ISO/IEC JTC 1/SC 22**). Software engineers are recommended to refer these standards as well.

5 Operating Systems

An operating system (OS) is software that manages the computer's hardware and provides a platform for software applications. Software engineers need a good general understanding of OSs and OS objectives, services, and functions.

Different types of OSs have been designed over time to support various types of systems or applications, including batch processing, multiprogramming, time-sharing, and dual-mode operation — for protecting I/O, memory, CPU, kernels and micro-kernels.

To choose an appropriate OS, software engineers have to analyze different types of operating systems, such as single-user, single-tasking, multiuser, multitasking and multi-threading OSs; real-time OS (RTOS); network OS; and distributed OS. For small systems, an operating system may not be required. It is important to study examples of each type and compare their benefits and limitations.

Software engineers need to understand operating systems' basic structure, system architecture types, design approaches, the architecture of distributed OS and issues in distributed OS.

An operating system typically has four major components: processor management, memory management, device management and information management.

5.1 Processor Management

Software engineers must understand the concepts of processor, process and address space. They must understand

booting, processes, cores, threads, user and kernel threads, fork and exec, synchronization, and hardware support for locking. They should compare and contrast various CPU scheduling concepts, scheduling algorithms, algorithm evaluations, multiple processor scheduling and real-time scheduling, concurrent programming, deadlocks, critical regions, conditional critical regions, and monitors.

Communication among different processes is important in multitasking, multiuser OSs. A software engineer must have a deep understanding of inter-process communication (IPC), and types of IPCs, including messages, pipes, shared memory, semaphores, modularization and process synchronization.

Various types of locks are used to ensure proper synchronization of data among processes, including semaphores, binary semaphores, counting semaphores and mutex locks. Deep understanding of common challenges of IPCs, deadlocks, deadlock scenarios, and deadlock characterization; prevention, avoidance, detection and recovery of deadlocks; and precedence graphs is critical and to be internalized with the help of case studies.

Software engineers are required to study, with examples, concurrent languages, processes and scheduling, job and process concepts, and various types of scheduling: CPU-I/O interleaving, non-preemption, context switching, and scheduling algorithms (first come, first served (FCFS), shortest job first (SJF), shortest remaining time first (SRTF), priority scheduling, round robin and combined schemes).

5.2 Memory Management

A software engineer needs a very good understanding of how memory is managed in the system and of the different types of memory and relevant concepts — physical memory, virtual memory, secondary memory, memory hierarchy, linking and memory allocation.

Engineers must understand memory fragmentation (both external fragmentation, internal fragmentation), and various memory management concepts, including units, paging, page tables, segmentation, paged segmentation, virtual memory management, demand paging, page replacement, thrashing and swapping.

Memory is allocated to processes in different ways — for example, through contiguous allocation, noncontiguous allocation, dynamic partitioned memory allocation, static-swapping and overlays.

An understanding of logical addresses, partitions, static versus dynamic memory allocation, free space management, and defragmentation of memory blocks is also important.

As the physical memory available is always limited, various memory page replacement strategies are designed and implemented. These strategies include First-in-first-out (FIFO), Not-Recently-Used (NRU), Least Recently Used (LRU), Most Recently Used (LRU), Least Frequently Used (LFU), Most Frequently Used (MFU), Longest Distance First (LDF), Second Chance, and Aging among others.

5.3 Device Management

A software engineer must have good knowledge of different types of I/O devices — memory-mapped and I/O-mapped devices, block and character

devices, and buffering devices. Engineers should compare and contrast polled, interrupt-driven and direct memory access (DMA) I/O devices, and blocking versus non-blocking I/O devices.

Device drivers are software programs that provide an interface between hardware and applications. Software engineers should understand device drivers, the various types of device drivers, device driver tables, device driver functions, and interfaces for various types of hardware devices, as well as hardware and software interrupts and interfaces by interrupts and polling.

Software engineers should also understand that issues with caching, scheduling, spooling and performance can arise for shared devices in multiuser, multitasking OSs and device a mechanism for resolving them.

5.4 Information Management

Software engineers need to understand the following:

- The concept of a process, a system programmer's view of processes, an operating system's view of processes, and operating system services for process management
- File system management, storage management, file attributes, directory structure, file system structure, mass storage structure, I/O systems, protection and security
- User and operating system views of the file system and various types of file systems — simple file system, symbolic file

system, logical file system and physical file system

Engineers should be familiar with various operations including access control lists (ACLs), access matrix, access control, access control verification, capabilities allocation strategy, I/O initiators, device strategy, device handlers, disk scheduling, disk space management, existence and concurrency control, schemes and combined schemes, authentication schemes, directory namespace, hierarchies, Directed Acrylic Graphs (DAGs), hard and soft links.

In addition, software engineers should understand important principles include hardware security, external security, operational security, password protection, access control, security kernels, and the layered approach.

5.5 Network Management

Network management is the process of administering and managing various types of networks. This content area includes network management concepts, distributed objects, distributed file systems, and network architecture, design, issues and resolutions.

A network manager will need detailed knowledge of physical and logical time, as well as internal and external synchronization protocols in network management such as Cristian's algorithm, Berkeley's algorithm, the Network Time Protocol, Lamport's logical clock, Vector clocks, Casual ordering of messages, and global state.

Other important topics include distributed computation, termination detection, distributed mutual exclusion and election, simple and multicast-based mutual exclusion algorithms; Centralized, Ring based, Ricart Agrawala's algorithm, Maekawa's algorithm, Election algorithms, Bully's algorithm and multicast communication.

6 Database Management

A database is a collection of related data elements, collected specifically for use by one or more applications and stored in an organized format for easy and quick access, using one or more key values. The data items or elements are stored in one or more databases or files, and the relationship among them is established using a database schema.

Basic operations performed on the database include creating the database and its elements (table, index, views, functions, procedures, etc.), deleting or dropping items from the database, modifying contents and structure of the database, and data retrieval, comment, and rename actions.

Different types of databases include relational databases, not only Structured Query Language (NoSQL) databases, columnar databases, object-oriented databases, key-value databases, document databases, hierarchical databases, graph databases, time series databases, and network databases. Understanding what type of database works best for specific applications and analyzing the definition, structure, specific pros and cons of each type of database; what along with examples helps software engineers choose the right type of database for a given application.

When selecting a database, software engineer should evaluate data models, storage models, types of databases, key values, graphs, column family, volume of data, consistent data access time, and the number of users or applications accessing the database (traffic), etc.

The learners and users of the database system need to create two roles (database user and database architect), review several case studies of

increasing complexity, create multiple databases, and analyze the information. This process significantly helps one to understand and internalize the database design and management.

6.1 Schema

A database schema is a structure or record of data items, defined in one or more database tables, and the relationships between them. The schema may also contain formulae to check the integrity of data items, relationships, indexes, functions or procedures and views.

While a physical schema explains how the database is designed at physical level (files), the logical schema describes how different data items are defined in one or more tables and interconnected.

Different types of schemata used in the industry include star, snowflake and fact constellation schemata. Different types of keys used in schemata include Primary Key, Secondary / Alternate Key, Foreign Key, Composite Key, Surrogate Key and Candidate Key.

Parameters that influence the definition and use of schemata include overlap preservation, extended overlap preservation, normalization and minimality.

6.2 Data Models and Storage Models

A data model specifies the logical aspects of data structure in a data store, and a storage model specifies the physical aspects of data structure in a data store. It is difficult to achieve both data consistency and high availability in a database.

The two primary data models used to distinguish databases are the following:

- i. The ACID (Atomicity, Consistency, Isolation, Durability) model provides for high data consistency. ACID-compliant databases are ideal for a finance-intensive application.
- ii. The BASE (Basically Available, Soft state, Eventual consistency) model provides flexible methods to process data, which suits NoSQL database types.

Types of storage models include the following:

- i. DAS (Direct Access Storage): Storage devices are physically or directly connected to the computer that processes the data.
- ii. NAS (network access storage): Data is stored in a network and accessed by multiple computers or applications.
- iii. SAN (storage area network): Data is stored in multiple servers and efficiently provided to users through a computer network.

6.3 Database Management Systems

Database Management Systems (DBMSs) are software systems that provide the necessary tools for maintaining data optimally, retrieving stored information effectively, protecting and securing stored data, and managing access for users of different levels of authority.

Typical DBMSs include:

- **A database engine:** This is the core of a DBMS. The database engine manages efficient storing and retrieving of data. Users with privileges can access the database engine.
- **A database manager:** This program or set of programs performs all DBMS functionality in a database (creating, purging, backing up, retrieving, maintaining, cloning and deleting data). It is also responsible for maintaining the DBMS with patches and updates.
- **A runtime database manager (RDM):** The RDM checks for user authentication and privileges before any operation is performed, provides access to a context-based database, provides concurrent access to the database by multiple users, and ensures data integrity.
- **Database languages:** These help in storing, retrieving, modifying and retrieving data, controlling user access (privileges), specifying schemata and views, and performing various operations. Popular database languages include Data Definition Language (DDL), Database Access Language (DAL), Data Manipulation Language (DML) and Data Control Languages (DCL)
- **A query processor:** This basic and key component of DBMS provides an effective, rich and English-like interface for users to access the database and

perform various functions or operations.

- **Reporting:** Reporting applies specified filters, extracts requested data and records from one or more database tables, and presents information as specified.

Several free and open-source database management systems are available.

6.4 Relational Database Management Systems and Normalization

Conventional file system-based databases suffered from data redundancy, data inconsistency, data access challenges, unauthorized access, lack of concurrent access, among other issues.

A Relational Database Management System (RDBMS) stores data in tables and, unlike in a DBMS, its data tables relate to one another, multiple data items can be accessed simultaneously, a large amount of data is handled, multiple users can access data concurrently, data redundancy is significantly reduced, and multiple levels of data security are supported.

Computer science engineers must understand the difference between the various types of RDBMS, such as Objective RDBMS, Object Oriented RDBMS, be familiar with examples, and know the applications they suit best.

Database normalization is the process of organizing data in a database and removing data redundancy and data inconsistency from the tables. Normalization might increase the number of tables and increase the

query time. If this occurs, then — depending on the application and the requirement — de-normalization is applied, where data redundancy is added for quicker data access.

Different types of database normalizations are the following:

- i. First Normal Form (1 NF): Removes duplication or redundancy. Each table cell has a single value (creates more entries and tables). Each row has unique values. Related data is identified with a unique key.
- ii. Second Normal Form (2 NF): The table should be in 1 NF; no partial dependency (creates separate tables with records referenced by multiple records or tables).
- iii. Third Normal Form (3 NF): The table should be in 2 NF. Transitive dependencies are removed.
- iv. Boyce-Codd Normal Form (BCNF/3.5 NF): The table should be in 3 NF, and X should be the super-key for any ($X \rightarrow Y$).
- v. Fourth Normal Form (4 NF): The table should be in 3.5 NF and should not have a multivalued dependency.
- vi. Fifth Normal Form (5 NF): The table should be in 4 NF and cannot be split into any more tables without losing data.
- vii. Sixth Normal Form or Domain/Key Normal Form (6 NF/DKNF): The table should be in 5 NF, and every join dependency is trivial.

Most databases are typically normalized until 3 NF or BCNF. An alternative normal form, DKNF, is defined where insertion and deletion of anomalies are avoided (see [x-Fagin]).

Database engineers are encouraged to understand normalization forms with examples and case studies and to understand the challenges one would face if the database were not normalized. Although normalization is essential and provides various benefits, it also increases the number of tables and processing time.

6.5 Structured Query Language

Structured Query Language (SQL) is a standard and popular database language for creating, updating, and deleting databases and for retrieving information from databases. SQL is an inevitable part of most database management systems.

Typical SQL syntax has several language constructs or elements, including clauses, expressions, predicates, queries and statements.

All operations on a database, including creating, updating, deleting and viewing tables; performing different normalizations; purging data; and searching through the database based on various combinations of parameters or filters, can be performed using SQL.

Most databases support SQL (except NoSQL databases), and the SQL syntax and library of functions supported vary across database providers (much like programming languages — though different languages support similar features, the syntaxes vary).

Database engineers also have to decide whether to use static/embedded SQL,

dynamic SQL or a combination of the two, after weighing the pros and cons of each option for the particular application. They should also know the differences between simple and complex views and use them appropriately.

SQL is standardized and adopted by the American National Standards Institute (ANSI) and ISO. The standards are revised from time to time; the first SQL standard was SQL-86, issued in 1986, and the most recent is SQL:2019.

6.6 Data Mining and Data Warehousing

Databases are designed to store transactions and retrieve them efficiently.

Data warehousing extracts data from multiple databases efficiently and stores it in a common database so data mining can be performed effectively on the compiled data. Data warehouses are typically huge, as they store historical data records.

Data mining extracts requested information from the data warehouse, applying various filters and conditions. Data mining applies pattern recognition algorithms to huge data sets to generate required reports.

The different types of warehouses include Enterprise Data Warehouse (EDW), Operational Data Store (ODS) and Data Mart (DM).

Many efficient tools are available to create data warehouses and mine data from them.

Database engineers must know different data mining techniques, including association, clustering,

classification, sequential patterns and prediction, and know how to apply them for various uses and industries, such as health care, fraud detection, customer relationship management, finance and banking, anomaly detection, prediction, neural networks, statistics, and data visualization.

6.7 Database Backup and Recovery

Database systems are prone to failures, and data can be corrupted. It is crucial to prevent data corruption and — if it does occur — to recognize it immediately and recover the data.

Updating the database for transactions must be carried out carefully (with commits at specific checkpoints), and must incorporate techniques such as undoing, deferred updates, immediate updates, caching or buffering, and shadow paging.

Databases must be backed up periodically to ensure data safety. Backup techniques include Full database backup, Differential backup and Transaction log backup.

7 Computer Networks and Communications

A computer network is a group of devices that are connected for sharing information. The connected devices (nodes on the network) can be located near one another, on the same premises, or somewhere else. Networking is required for certain benefits, including certain modes of communication and information sharing; the ability to share devices such as printers, routers and video cameras; global information and data storing; security and policy enforcement; remote monitoring; shared business models; and web browsing.

As we are in the internet era, computer networking is a critical element in computing, and the practitioners of computer science engineering have to study computer networks and communication concepts, including examples and case studies. Many computing paradigms (distributed computing, grid computing, cloud computing, etc.) are based on networking principles.

It is important for software engineers to understand the following:

- Different types of computer networks
- Layered architectures of networks
- Open Systems Interconnect (OSI) layers
- Encapsulation and decapsulation
- Application layer protocols
- Design techniques for reliable and efficient networking

- Internet and packet delivery
- Wireless and mobile networks
- Security and vulnerabilities

7.1 Types of Computer Networks

Different types of computer networks are designed and used based on the need, such as the following:

1. Personal Area Network (PAN) / Home Network
2. Local Area Network (LAN)
3. Wireless Local Area Network (WLAN)
4. Wide Area Network (WAN)
5. Campus Area Network (CAN)
6. Metropolitan Area Network (MAN)
7. Storage Area Network (SAN)
8. System-Area Network (SAN)
9. Enterprise Private Network (EPN)
10. Virtual Private Network (VPN)

It is important to understand each of the above network type as well as examples, benefits, limitations and available solutions to circumvent challenges.

7.2 Layered Architectures of Networks

A communication system includes hardware and software, and these components have become complex to meet complicated use scenarios and user demands. To support the implementation and maintenance of such systems, ISO has developed a layered approach, where every layer

has specific functionality for processing data and transferring it from one node to another.

Each layer is independent in its functionality and provides services from the lower layer to the upper layer without providing details of how each layer's service is implemented. Each layer ("n") on a machine communicates with the same layer ("n") on the peer machine. Rules used in a conversation are called *layer-n protocol*.

The basic elements of the layered approach are service, protocol and interface.

- **Service:** The set of actions a layer provides to the adjacent higher layer is the service.
- **Protocol:** The set of rules a layer uses to exchange information with the peer entity is called the protocol. The rules are primarily for managing both the contents and order of the messages used.
- **Interface:** The interface provides a medium for transferring the message from one layer to another layer.

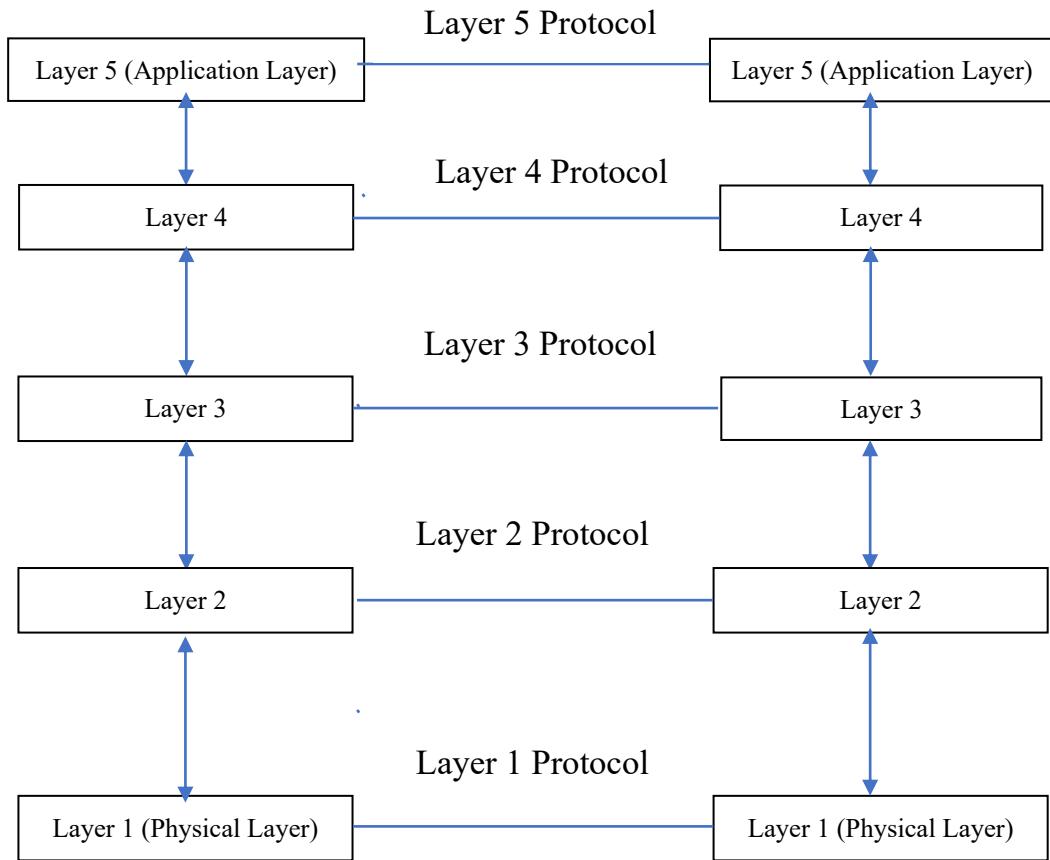


Figure 16.4: Pictorial Representation of Layered Networking

Software engineers are expected to understand the essential functionalities required, various modes in which the data or information is communicated from one layer to the other, and data packet formation and interpretation at peer levels. A useful exercise is to take examples of different protocols and analyze them.

7.3 Open Systems Interconnection Model

The Open Systems Interconnection (OSI) Model was defined by the ISO. It serves as a reference model for information exchange between applications on two systems or computers through a physical medium.

OSI proposes seven (7) layers, and each layer is assigned a specific task. Each layer independently processes the data it receives from the upper or lower layer and passes it to the lower or upper layer, as appropriate.

Engineers must understand each OSI layer, its functionality protocol, the input and output of each layer in each direction (from lower layer to upper layer and vice versa). Engineers should analyze whether all seven layers are required for all protocols and what is necessary to optimize for performance.

1. Physical Layer (Layer 1)
2. Data Link Layer (Layer 2)
3. Network Layer (Layer 3)

4. Transport Layer (Layer 4)
5. Session Layer (Layer 5)
6. Presentation Layer (Layer 6)
7. Application Layer (Layer 7)

Engineers must understand the nuances of each layer, with examples.

7.4 Encapsulation and Decapsulation

Each layer, while sending data from the upper layer to the lower layer, inserts additional information at the beginning (header) and optionally at the end of the data packet received from the upper layer, treating the packet received from the upper layer as data. This is encapsulation. The Protocol Data Unit (PDU), which is the data packet containing additional information from all layers, is sent to the receiving system. At the receiving end, each layer extracts its header from the PDU, deciphers the information to treat the data appropriately, and sends the remaining PDU to the upper layer.

Learning about cross-layer optimization, the principles to which it must adhere, and its applications is important. Engineers should analyze the PDU structures of each layer of OSI, the Internet protocol suite and the Asynchronous Transfer Mode (ATM).

7.5 Application Layer Protocols

The application layer, being the top most layer, provides services and interfaces to interact with users' application. There are two types of application layers in the OSI model: Common Application Service Element (CASE) and Specific Application Service Element (SASE). Example applications include File Transfer (FTP, TFTP, NFS), Remote login

(Telnet, Zoho Assist, Anydesk, TeamViewer, etc), e-mail (SMTP) Networking Support (DNS), Network Management (SNMP, DHCP), Devices (LPD), etc.

Software engineers practicing in a networking domain need to understand CASE and SASE application services, including example applications in each category.

7.6 Design Techniques for Reliable and Efficient Network

Today's information technology-based businesses need around-the-clock, reliable, efficient and scalable networks and high-speed internet availability. Catering to varied business needs, the networks and their management has become complex as well.

It is critical to identify network requirements (both business goals and technical solutions) along with a road map (scalability). The fundamental design goals should include reliability, security, availability and manageability. Engineers should expect threats and intrusions at multiple levels and design security at multiple levels. Systems must be set up to monitor the networks for both proper functioning and malfunctioning; identify faults, vulnerabilities and hacks quickly; and fix them.

Engineers must understand and learn the nuances of designing a network while using appropriate Firewalls, LAN/VLANs, Subnets, Quality of Service (QoS), Demilitarized Zone (DMZ), Spanning Tree (especially for hierarchical network), Port or network interface controller (NIC) channel, security (both poll security and physical security), wireless access points, and wireless access controllers.

Even when the design and implementation are well planned and executed, one has to be constantly vigilant for attacks and continuously upgrade to better systems, devices and tools.

7.7 Internet Protocol Suite

Data is transmitted in packets from one computer to another, either in the same network or in a different one. The Internet Protocol suite, or TCP/IP, defines data communication between two computers connected via the internet. The top three layers of the OSI model (Application, Presentation and Session layers) are merged into the application layer, and the network layer is revised specifically for internet functioning. Internet Protocol is the fulcrum of today's internet or network layer.

Multiple variations of Internet Protocols are designed and used for different purposes. The protocols include TCP/IP (Transmission Control Protocol/Internet Protocol), UDP/IP (User Datagram protocol / Internet Protocol), SMTP (Simple Mail Transfer Protocol), PPP (Point to Point Protocol), FTP File Transfer Protocol, SFTP (Secure FTP), HTTP (Hyper Text Transfer Protocol), HTTPS (HTTP Secure), Telnet (Terminal Network), POP3 (Post office Protocol 3), VOIP (Voice over Internet Protocol), SLIP (Serial Line Internet Protocol), PPP (Point-to-Point Protocol). It is important to know the differences between these along with use cases (applications where each type is used or where it works best).

Mobile Internet Protocol is a communications protocol that conforms to an IETF (Internet Engineering Task Force) standard and allows users to move their mobile

devices (laptops, mobile phones, etc.) seamlessly from one network to the other without changing the IP address.

Internet Protocol Version 4 (IPV4) uses a 32-bit IP address, whereas IPV6 uses 128-bit IP addresses.

Private IP addresses are translated into public IP addresses using either NAT (Network Address Translation) or PAT (Port Address Translation). Both use IPV4, but PAT uses port numbers. Different technologies used to communicate between IPV4 and IPV6 devices include dual-stack routers, tunneling and NAT protocol translators.

Professional computer network architects and programmers need to understand IPV6 addressing, routing, transitioning to IPV6 from IPV4, dual-dress stacks, tunneling and NAT64.

7.8 Wireless and Mobile Networks

Wireless networks provide the ability for devices to connect and communicate without the hassle of wires and cables. They also provide flexibility and ease of using the devices. Different wireless technologies are used for different applications:

- Wireless Personal Area Networks (WPAN)
- Wireless Local Area Networks (WLAN)
- Wireless Wide Area Networks (WWAN)

A mobile or cellular network is a radio network spread over a specific area of land (called a cell). The cells are served by base stations, which are fixed-

location transceivers. To avoid interference and ensure guaranteed bandwidth, the adjacent cells use a different set of frequencies. These cells, when connected, provide wide area radio coverage. The cell patterns take different shapes, but squares, circles and hexagons are typical.

Different methods of data transmission are used between channels, such as Frequency Division Multiple Access (FDMA), Time Division Multiple Access (TDMA), Code Division Multiple Access (CDMA), Space Division Multiple Access (SDMA), etc.

Wireless technology has evolved over several generations. Software Engineers are encouraged to learn the differences among 1G, 2G, 3G, 4G and 5G technologies, along with the core network, access system, frequency, bandwidth and technologies used in each.

7.9 Security and Vulnerabilities

Although wireless technology provides the ease of connecting seamlessly to the network, it is also prone to attacks unless the network is secured. Risks to unsecured wireless networks include Piggybacking, Wardriving, Evil Twins attacks, Wireless sniffing, Unauthorized computer access, Shoulder sniffing and Theft of mobile devices.

Communication over the internet via mobile device is highly vulnerable to cyberattacks. In addition to wardriving, mentioned above, typical wireless and mobile device attacks include SMiShing, War driving, WEP attacks, WPA attacks, Bluejacking, Reply attacks, Blue snarfing, RF Jamming, etc.

Many precautionary measures must be implemented and strictly followed to reduce such risks. These measures include changing default passwords, changing passwords frequently, restricting access to authorized users, encrypting data in the system and on the network, and installing multiple levels of firewalls. In addition, users must protect and hide (not publicize) Service Set Identifier (SSID), use effective antivirus software, and update and upgrade it regularly; use a Virtual Private Networks (VPN), use file-sharing or system-sharing access with care, and disable access after use; and update or upgrade the access point or access controller, gateway and other devices with security patches when they become available.

8 User and Developer Human Factors

The thought processes and behaviors of software developers typically differ from that of software users. This content area identifies salient parameters that matter for end users as well as the perspective of the developers. Human-computer interface (HCI) focuses on designing and developing computer technology for users to interact with computing systems.

User satisfaction is measured in terms of User Experience (UX). An ideal interface would facilitate interaction that is as natural as the interaction between two human beings.

8.1 User Human Factors

Users expect software to be robust; to have an intuitive graphical user interface (GUI) that guides the user through minimal, intelligent, easy-to-follow steps to achieve the end result; to be secure; and to provide fast, consistent responses.

The interface should help users use the system easily. The interface should be self-explanatory and enable self-learning. The messages, whether communicating results or errors, should be clear and complete. The system should be able to regain its original state if there are errors.

The system should allow users to interrupt during the processing and undo the operation, wherever possible.

The software engineer needs to identify the profile of users the system; system's functionality, input and output interfaces users use (keyboard, touch pad, audio, video, etc.) to interact

with the system, the system's fault tolerance, the system's performance parameters. among others.

Typically, user interface development goes through several iterations, starting with a prototype. The user interface devices must be robust.

8.2 Developer Human Factors

The software lives much longer than the time taken to develop. Invariably, the software engineers who maintain the code are different from those who develop. Hence, the code has to be written with more care and for use by other programmer / software engineer.

Meaningful and comprehensive documentation is crucial at all stages of software lifecycle.

Defining and adopting apt coding standard for the project, and ensuring every team member implements the same in spirit is key for developing clean code that lives longer with minimal maintenance.

Programming style is another key ingredient of a good code. Code has to be legible, should be like reading a good poem and easily comprehensible. Using meaningful, consistent and detailed comments is essential to ensure code readability.

Other traits of a good software programmer include being a team player, enjoy solving puzzles creatively, be agile, be structured / modular among others.

Good coding standards include defining naming conventions for various types of variables, functions / procedures, comment structure / styles, indentation styles, structuring the code

into paragraphs (of related functions),
etc.

“Code is read many more times than it
is written. Consider whether write-
time convenience is a false economy” -
Steve McConnell

“Clean code always looks like it was
written by someone who cares” -
Robert (Uncle Bob) Martin

9 Artificial Intelligence and Machine Learning

Intelligence is the ability to acquire and correlate information and knowledge to make a correct decision for a specific task. *Artificial intelligence (AI)* enables computer systems to become intelligent, like human beings. *Machine learning (ML)* enables computer systems to learn from experiences and to use the knowledge gained to make smart decisions — to become artificially intelligent. Deep learning uses artificial neural network models for learning and making predictions.

Everyone expects all systems they use to be smart, reliable, consistent, secure and fault-tolerant — and to get better every day. AI and ML work toward enabling systems to accomplish all this.

An ideal AI system would be one that a human could not identify it as a computer; humans would not be able to distinguish the computer from a human being.

Several tools have been developed and are available for creating AI systems. Using proven tools helps engineers build a stable system faster.

9.1 Reasoning

Reasoning means analyzing sets of information available for a given situation and determining the cause of the situation. Reaching this conclusion is an important ability of AI, as the conclusion informs AI's decision about what to do next.

Different types of reasoning used in AI include the following:

Deductive Reasoning is a standard and strategic approach to mapping available facts, information and knowledge to arrive at a conclusion. In this approach, available facts and information are considered to be authentic. For example, if the premises are “All girls are beautiful” and “Michu is a girl,” then the conclusion is “Michu is beautiful.”

Inductive Reasoning is about introducing a hypothesis and creating generalizations from the available facts and premises. Unlike deductive reasoning, in inductive reasoning, even if the premises are certain, the conclusion would be *probable*, depending on whether the inductive argument is strong or weak. For example, check the location of all engineers working on a project and if they are from Bengaluru, India state “All employees working on the gaming project are from Bengaluru.”

Abductive Reasoning starts with an incomplete set of data or information and proceeds to derive the most likely conclusion from the latest data. For example, a doctor analyzes the latest lab reports of a patient to predict the course of the disease.

Common Sense Reasoning makes inferences about situations based on similar past experiences. For example, if a motorcycle skids while driving on a wet road, that information is remembered and considered during future rides.

Monotonic Reasoning occurs when the conclusion remains permanent or constant after it is reached. For example, “The Himalayas are one of the tallest mountain ranges.”

Non-Monotonic Reasoning

(NMR) occurs when the inference changes values or direction based on new knowledge or information. NMR is based on assumptions and deals with incomplete or not-known facts. For example, the rule is “Birds fly”. But a few birds do not fly including penguins.

Software engineers are encouraged to learn other reasoning methods, such as metalevel reasoning, procedural numeric reasoning, and formal reasoning, as well.

9.2 Learning

We learn from our observations, experiments and experiences. Enabling computers to learn and to remember what they've learned for future use is critical for building AI systems. An AI system learns when observations and outcomes of experiments (signals) are fed back into the system. Different types of learning include the following:

Supervised Learning, the computer system trains by receiving labeled (i.e., training) data. Subsequently, when any input is provided, the system compares it with the data it was trained on and generates output. Naturally, the more training data, the better the outcome. Supervised learning uses multiple learning techniques, including the classification technique and the regression technique. Supervised learning may not be able to handle complex tasks.

Unsupervised Learning, labeled or training data is not provided to the system. The system has to figure out common patterns from the input given and make inferences. The data is analyzed in real time.

Semi-supervised Learning, the system is trained with partly labeled and partly unlabeled data. This type of learning has been shown to be effective.

Reinforcement Learning is based on interactions with the environment. In this type of learning, the system receives feedback (an error message or a reward) and learns from that feedback. No data is provided to the system (neither labeled nor unlabeled). Various algorithms are produced in reinforced learning. This is a trial-and-error method for learning.

Software engineers working on AI are expected to know various other learning techniques as well, including dimensionality reduction learning, self-learning, feature learning, sparse learning, anomaly detection and robot learning, along with the key differences between the methods and the applications where each method works well.

9.3 Models

AI models are inference engines or tools (algorithms) that can arrive at the best decisions based on relevant data.

Different models are created to enable efficient ML, with or without training data. Models used in ML include the following:

Linear Regression model is based on supervised learning, where the relationship between input and output variables is determined and used. This model is commonly used in health care and banking applications.

Logistic Regression model is a statistical model primarily used for

classifying dependent variables from given independent variables.

Artificial Neural Networks are inspired by biological neural networks in a brain. The systems are designed to learn naturally from the inputs without specific rules.

Decision Tree model is used where past decisions are used to arrive at a decision. The name “tree” is used because the data is stored in the form of a tree.

Naïve Bayes model works on the assumption that the presence of a feature does not depend on the presence of any other feature. Spam filtering is one of the applications that suits this model.

Support Vector Machine (SVM), is a supervised ML algorithm used to analyze a limited quantum of data. SVM is typically faster than artificial neural networks because it works with limited data.

Random Forest model uses multiple decision trees for making a final decision. The random forest model is useful for solving both regression and classification problems.

AI models are key to making the most appropriate decisions. As different models suit specific applications or domains, software engineers are encouraged to learn many other AI models as well, such as Linear Discriminant Analysis, Learning Vector Quantization, K-nearest Neighbors (KNN), etc.

9.4 Perception and Problem-Solving

Solving a problem efficiently and quickly is the goal of AI. Problem-solving predominantly comprises understanding user commands and executing them, as humans do. Depending on the application and problem to be solved, AI systems use the relevant knowledge base and predicate logic to identify the most appropriate solution.

AI systems dealing with the external world, obtain environmental data through sensors (cameras; microphones; temperature, pressure and light sensors, etc.), analyzes the data using its knowledge base or inference engine, and acts upon it.

Based on capabilities and functionality, AI systems are categorized into multiple types.

Type I AI systems are designed to do specific tasks with intelligence. Examples include Chess games, speech and image recognition, among others.

Type II AI systems analyze the current situation or environment and do not normally refer to previous decisions made in a similar situation to arrive at an appropriate action. Reactive systems or reactive machines typically make decisions and execute commands at that instance, referring to the existing knowledge base. A good example is a self-driving cars.

Type III, or self-aware, AI systems have consciousness and are mindful. These systems adopt the mind theory and predict the mood of the other person or entity based on the person’s action or type of action. For example, if the driver in the vehicle behind the system honks, then the AI system might conclude that the driver is angry or unhappy. Social and ethical behavior is part of conscious systems.

9.5 Natural Language Processing

Natural language processing (NLP) is a crucial part of AI systems, enabling users to interact with the AI systems in a way that is similar to how they interact with other humans. AI systems understand human languages and execute commands delivered in those languages. AI systems that work on voice commands need to understand not only the human language, but also the slang or pronunciation of the user.

9.6 AI and Software Engineering

Software engineering and AI are mutually related to each other in basically two ways: AI applications in software engineering (i.e., AI for SE) and software engineering for AI systems (i.e., SE for AI).

AI for SE aims to establish efficient ways of building high-quality software systems by replicating human developers' behavior. It ranges over almost all development stages, from resolving ambiguous requirements to predicting maintainability, particularly well applied in software quality assurance and analytics, such as defect prediction, test case generation, vulnerability analysis, and process assessment [15]. Although human-centric software engineering activities benefit, engineers should be aware of limitations and challenges inherent to the nature of AI and ML, especially the uncertain and stochastic behavior and the necessity of sufficiently labeled and structured datasets [15].

The development of AI systems is different from traditional software systems since the rules and system behavior of AI systems are inferred

from training data rather than written down as program code [16]. Thus, there is a need for particular support of SE for AI, such as interdisciplinary collaborative teams of data scientists and software engineers, software evolution focusing on large and changing datasets, and ethics and equity requirements engineering [16]. Recommended software engineering practices for AI are often formalized as patterns, such as ML software design patterns [17].

REFERENCES

- [1] *Joint Task Force on Computing Curricula*, IEEE Computer Society and Association for Computing Machinery, Software Engineering 2014: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering, 2014; <http://sites.computer.org/ccse/SE2004Volume.pdf>.
- [2*] G. Voland, *Engineering by Design*, 2nd ed., Prentice Hall, 2003.
- [3*] S. McConnell, *Code Complete*, 2nd ed., Microsoft Press, 2004.
- [4*] J.G. Brookshear, *Computer Science: An Overview*, 12th ed., Addison-Wesley, 2017.
- [5*] E. Horowitz et al., *Computer Algorithms*, 2nd ed., Silicon Press, 2007.
- [6*] I. Sommerville, *Software Engineering*, 9th ed., Addison-Wesley, 2011.
- [7] ISO/IEC/IEEE 24765:2017 *Systems and Software Engineering—Vocabulary*, ISO/IEC/IEEE, 2010.
- [8*] L. Null and J. Lobur, *The Essentials of Computer Organization and Architecture*, 5th ed., Jones and Bartlett Publishers, 2018.
- [9*] J. Nielsen, *Usability Engineering*, Morgan Kaufmann, 1994.
- [10] ISO 9241-420:2011 *Ergonomics of Human-System Interaction*, ISO, 2011.
- [11*] M. Bishop, *Computer Security: Art and Science*, 2nd ed Addison-Wesley, 2018.
- [12] R.C. Seacord, *The CERT C Secure Coding Standard*, Addison-Wesley Professional, 2016.
- [13] R. Fagin, “A Normal Form for Relational Databases that is based on Domains and Keys,” *ACM Transactions on Database Systems*, Vol. 6, No. 3, ACM, September, 1981
- [14] I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning (Adaptive Computation and Machine Learning series)* Illustrated Edition, illustrated edition, 2018
- [15] S. Shafiq, A. Mashkoor, C. Mayr-Dorn, A. Egyed, “A Literature Review of Using Machine Learning in Software Development Life Cycle Stages,” *IEEE Access*, Volume 9, IEEE, October, 2021
- [16] S. Martínez-Fernández, J. Bogner, X. Franch, M. Oriol, J. Siebert, A. Trendowicz, A. M. Vollmer, “Software Engineering for AI-Based Systems: A Survey,” *ACM Transactions on Software Engineering and Methodology*, Vol. 31, No. 2, ACM, April, 2022
- [17] H. Washizaki, F. Khomh, Y. G. Gueheneuc, H. Takeuchi, N. Natori, T. Doi, S. Okuda, “Software Engineering Design Patterns for Machine Learning Applications,” *Computer*, Vol. 55, No. 3, IEEE Computer Society, March, 2022

CHAPTER 17

MATHEMATICAL FOUNDATIONS

ACRONYMS

CFG	Context Free Grammar
CSG	Context Sensitive Grammar
FSM	Finite State Machine
GCD	Greatest Common Divisor
LHS	Left-Hand Side
PSG	Phrase Structure Grammar
RHS	Right-Hand Side

INTRODUCTION

Software engineers live with programs. Simply stated, one can program only for something that follows well-understood, non-ambiguous logic. The Mathematical Foundations knowledge area (KA) helps software engineers comprehend this logic, which in turn is translated into source code. The mathematics that is the primary focus in this KA is quite different from typical arithmetic, where numbers are dealt with and discussed. Logic and reasoning are the essence of mathematics that a software engineer must address.

Mathematics, in a sense, is the study of formal systems. The word “formal” is associated with precision, so there cannot be any ambiguous or erroneous interpretation of the fact. Mathematics is therefore the study of any and all certain truths about any concept. This concept can be about numbers as well as about

symbols, images, sounds, video—almost anything. In short, numbers and numeric equations aren’t the only subjects to precision. On the contrary, a software engineer needs to have a precise abstraction on a diverse application domain.

The Mathematical Foundations KA covers basic techniques to identify a set of rules for reasoning in the context of the system under study. Anything that one can deduce following these rules is an absolute certainty within the context of that system. In this KA, techniques that can represent and take forward the reasoning and judgment of a software engineer in a precise (and therefore mathematical) manner are defined and discussed. The language and methods of logic that are discussed allow software engineers to describe mathematical proofs to infer conclusively the absolute truth of certain concepts beyond just numbers. The objective of this KA is to help software engineers develop the skill to identify and describe such logic as well as verify that the logic in the code is consistent with abstractions. The emphasis is on helping software engineers understand the basic concepts rather than on challenging arithmetic abilities.

BREAKDOWN OF TOPICS FOR MATHEMATICAL FOUNDATIONS

The breakdown of topics for the Mathematical Foundations KA is shown in Figure tbd.1.

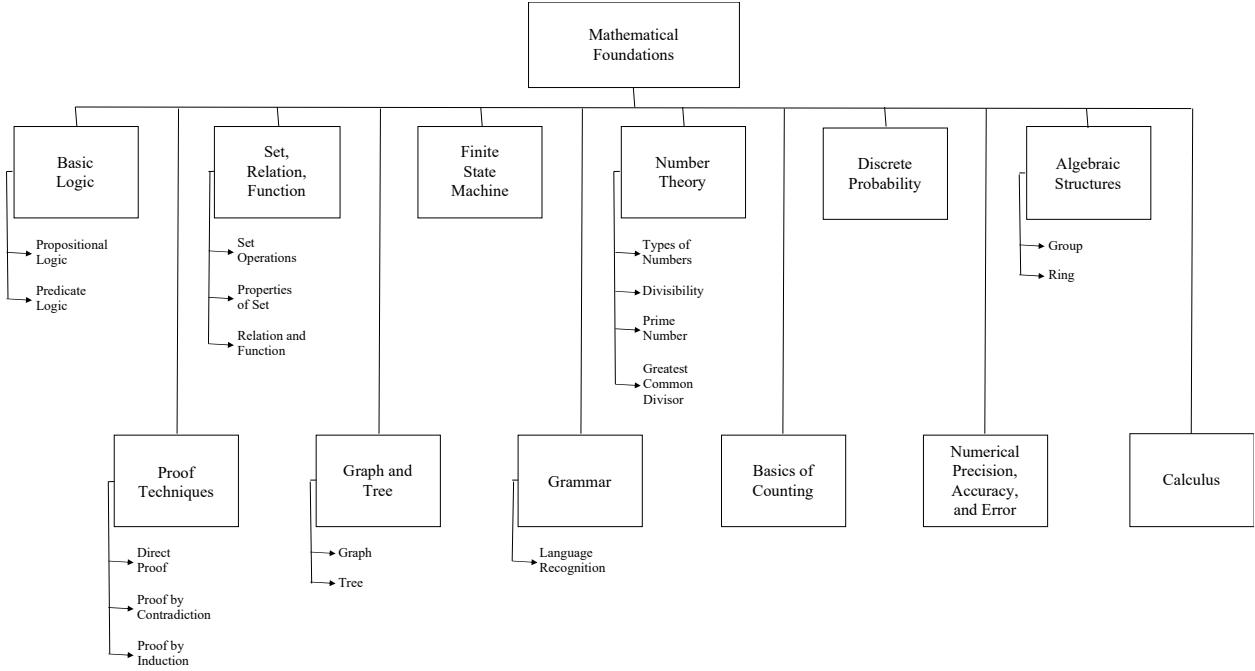


Figure tbd.1. Breakdown of Topics for the Mathematical Foundations KA

1. Basic Logic

[1*, c1]

1.1. Propositional Logic

A proposition is a statement that is either true or false, but not both. Consider declarative sentences for which it is meaningful to assign either of the two status values: *true* or *false*. Some examples of propositions are given below.

1. The sun is a star
2. Elephants are mammals.
3. $2 + 3 = 5$.

However, $a + 3 = b$ is not a proposition, as it is neither true nor false. It depends on the values of the variables a and b .

The Law of Excluded Middle: For every proposition p , either p is true, or $\neg p$ is false.

The Law of Contradiction: For every proposition p , it is not the case that p is both true and false.

Propositional logic is the area of logic that deals with propositions. A truth table displays the relationships between the truth values of propositions.

A Boolean variable is one whose value is either true or false. Computer bit operations correspond to logical operations of Boolean variables.

The basic logical operators include negation (\neg , $\neg p$), conjunction (\wedge , $p \wedge q$), disjunction (\vee , $p \vee q$), exclusive or (\oplus , $p \oplus q$), and implication (\rightarrow , $p \rightarrow q$). Compound propositions may be formed using various logical operators.

A compound proposition that is always true is a tautology. A compound proposition that is always false is a contradiction. A compound proposition that is neither a tautology nor a contradiction is a contingency.

Compound propositions that always have the same truth value are called logically

equivalent (denoted by \equiv). Some of the common equivalences are:

Identity laws:

$$p \wedge T \equiv p \quad p \vee F \equiv p$$

Domination laws:

$$p \vee T \equiv T \quad p \wedge F \equiv F$$

Idempotent laws:

$$p \vee p \equiv p \quad p \wedge p \equiv p$$

Double negation law:

$$\neg(\neg p) \equiv p$$

Commutative laws:

$$p \vee q \equiv q \vee p \quad p \wedge q \equiv q \wedge p$$

Associative laws:

$$(p \vee q) \vee r \equiv p \vee (q \vee r) \quad (p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$$

Distributive laws:

$$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$$

$$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$$

De Morgan's laws:

$$\neg(p \wedge q) \equiv \neg p \vee \neg q \quad \neg(p \vee q) \equiv \neg p \wedge \neg q$$

1.2. Predicate logic

A predicate is a verb phrase template that describes a property of objects or a relationship among objects represented by the variables. For example, in the sentence, *The flower is red*, the template *is red* is a predicate. It describes the property of a flower. The same predicate may be used in other sentences.

Predicates are often given a name, e.g., "Red" or simply "R" and can be used to represent the predicate *is red*. Assuming R as the name for the predicate *is red*, sentences that assert an object is colored red can be represented as $R(x)$, where x represents an arbitrary object. $R(x)$ reads as *x is red*.

Quantifiers allow statements about entire collections of objects rather than having to enumerate the objects by name.

The Universal quantifier $\forall x$ asserts that a sentence is true for all values of variable x . For example, $\forall x \text{ Tiger}(x) \rightarrow \text{Mammal}(x)$ means all tigers are mammals.

The Existential quantifier $\exists x$ asserts that a sentence is true for at least one value of variable x . For example, $\exists x \text{ Tiger}(x) \rightarrow \text{Man-eater}(x)$ means there exists at least one tiger that is a man-eater.

Thus, while universal quantification uses implication, the existential quantification naturally uses conjunction.

A variable x that is introduced into a logical expression by a quantifier is bound to the closest enclosing quantifier. Similarly, in a block-structured programming language, a variable in a logical expression refers to the closest quantifier within whose scope it appears. For example, in $\exists x (\text{Cat}(x) \wedge \forall x (\text{Black}(x)))$, x in $\text{Black}(x)$ is universally quantified. The expression implies that cats exist, and everything is black.

A variable is said to be a free variable if it is not bound to a quantifier.

Propositional logic falls short in representing many assertions that are used in mathematics, computer science and therefore software engineering. It also fails to compare equivalence and some other types of relationship between propositions. For example, the assertion *a is greater than 1* is not a proposition because one cannot infer whether it is true or false without knowing the value of a . Thus, propositional logic cannot deal with such sentences. However, such assertions appear quite often in mathematics, and we want to infer on those assertions. Also, the pattern involved in the following two logical equivalences cannot be captured by propositional logic: "*Not all men are smokers*" and "*Some men don't smoke*." Each

of these two propositions is treated independently in propositional logic. There is no mechanism in propositional logic to determine if the two are equivalent to one another. Hence, in propositional logic, each equivalent proposition is treated individually rather than dealing with a general formula that covers all equivalences collectively.

Predicate logic is supposed to be a more powerful logic that addresses these issues. In a sense, predicate logic (also known as first-order logic or predicate calculus) is an extension of propositional logic to formulas involving terms and predicates.

2. Proof Techniques

[1*, c1]

A proof is an argument that rigorously establishes the truth of a statement. Proofs can themselves be represented formally as discrete structures.

Statements used in a proof include axioms and postulates that are essentially the underlying assumptions about mathematical structures, the hypotheses of the theorem to be proved, and previously proved theorems.

A theorem is a statement that can be shown to be true.

A lemma is a simple theorem used in the proof of other theorems.

A corollary is a proposition that can be established directly from a theorem that has been proved.

A conjecture is a statement whose truth value is unknown.

When a conjecture's proof is found that conjecture becomes a theorem. Many times, conjectures are shown to be false and, hence, are not theorems.

2.1. Direct Proof

Direct proof is a technique to establish that the implication $p \rightarrow q$ is true by showing that q must be true when p is true. For example, to show that if n is odd then $n^2 - 1$ is even, suppose n is odd, i.e., $n = 2k + 1$ for some integer k :

$$\therefore n^2 = (2k + 1)^2 = 4k^2 + 4k + 1.$$

As the first two terms of the Right-Hand Side (RHS) are even numbers irrespective of the value of k , the Left-Hand Side (LHS) (i.e., n^2) is an odd number. Therefore, $n^2 - 1$ is even. Direct proof can also be called Proof by Deduction.

2.2. Proof by Contradiction

A proposition p is true by contradiction if proved based on the truth of the implication $\neg p \rightarrow q$ where q is a contradiction. For example, to show that the sum of $2x + 1$ and $2y - 1$ is even, assume that the sum of $2x + 1$ and $2y - 1$ is odd. In other words, $2(x + y)$, which is a multiple of 2, is odd. This is a contradiction. Hence, the sum of $2x + 1$ and $2y - 1$ is even.

An inference rule is a pattern establishing that if a set of premises are all true, then it can be deduced that a certain conclusion statement is true. The reference rules of addition, simplification, and conjunction need to be studied.

2.3. Proof by Induction

Proof by induction is done in two parts. First, the proposition is established to be true for a base case—typically for the positive integer 1. In the second part, it is established that if the proposition holds for an arbitrary positive integer k , then it must also hold for the next greater integer, $k + 1$. In other words, proof by induction is based on the rule of inference that tells us that the truth of an infinite sequence of propositions $P(n)$, $\forall n \in [1 \dots \infty]$ is established if $P(1)$ is true, and secondly, $\forall k$

$\in [2 \dots n]$ if $P(k) \rightarrow P(k + 1)$.

For a proof by induction, it is not assumed that $P(k)$ is true for all positive integers k . Proving a theorem or proposition only requires us to establish that if it is assumed $P(k)$ is true for any arbitrary positive integer k , then $P(k + 1)$ is also true. The correctness of induction as a valid proof technique is beyond discussion in this KA. The following proposition is proved using induction.

Proposition: *The sum of the first n positive odd integers $P(n)$ is n^2 .*

Basis Step: The proposition is true for $n = 1$ as $P(1) = 1^2 = 1$. The basis step is complete.

Inductive Step: The induction hypothesis (IH) is that the proposition is true for $n = k$, k being an arbitrary positive integer k .

$$\therefore 1 + 3 + 5 + \dots + (2k - 1) = k^2$$

Now, it's to be shown that $P(k) \rightarrow P(k + 1)$.

$$\begin{aligned} P(k + 1) &= 1 + 3 + 5 + \dots + (2k - 1) + (2k + 1) \\ &= P(k) + (2k + 1) \\ &= k^2 + (2k + 1) \text{ [using IH]} \\ &= k^2 + 2k + 1 \\ &= (k + 1)^2 \end{aligned}$$

Thus, it is shown that if the proposition is true for $n = k$, then it is also true for $n = k + 1$.

The basis step together with the inductive step of the proof show that $P(1)$ is true and the conditional statement $P(k) \rightarrow P(k + 1)$ is true for all positive integers k . Hence, the proposition is proved.

3. Set, Relation, Function

[1*, c2]

Set. A set is a collection of objects, called elements. A set can be represented by listing its elements between braces, e.g., $S = \{1, 2, 3\}$.

The symbol \in is used to express that an

element belongs to a set, or—in other words—is a member of the set. Its negation is represented by \notin , e.g., $1 \in S$, but $4 \notin S$.

In a more compact representation of set using set builder notation, $\{x \mid P(x)\}$ is the set of all x such that $P(x)$ for any proposition $P(x)$ over any universe of discourse. Examples for some important sets include the following:

$N = \{0, 1, 2, 3, \dots\}$ = the set of nonnegative integers.

$Z = \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$ = the set of integers.

Finite and Infinite Set. A set with a finite number of elements is called a finite set. Conversely, any set that does not have a finite number of elements in it is an *infinite set*. The set of all natural numbers, for example, is an infinite set.

Cardinality. The cardinality of a finite set S is the number of elements in S . This is represented $|S|$, e.g., if $S = \{1, 2, 3\}$, then $|S| = 3$.

Universal Set. In general $S = \{x \in U \mid p(x)\}$, where U is the universe of discourse in which the predicate $P(x)$ must be interpreted. The "universe of discourse" for a given predicate is often referred to as the universal set. Alternately, one may define universal set as the set of all elements.

Set Equality. Two sets are equal if and only if they have the same elements, i.e.:

$$X = Y \equiv \forall p (p \in X \leftrightarrow p \in Y).$$

Subset. X is a subset of set Y , or X is contained in Y , if all elements of X are included in Y . This is denoted by $X \subseteq Y$. In other words, $X \subseteq Y$ if and only if $\forall p (p \in X \rightarrow p \in Y)$. If $X = \{1, 2, 3\}$ and $Y = \{1, 2, 3, 4, 5\}$, then $X \subseteq Y$.

If X is not a subset of Y , it is denoted as $X \not\subseteq Y$.

Proper Subset. X is a proper subset of Y (denoted by $X \subset Y$) if X is a subset of Y but not equal to Y, i.e., there is some element in Y that is not in X.

In other words, $X \subset Y$ if $(X \subseteq Y) \wedge (X \neq Y)$.

If $X = \{1, 2, 3\}$, $Y = \{1, 2, 3, 4\}$, and $Z = \{1, 2, 3\}$, then $X \subset Y$, but X is not a proper subset of Z. Sets X and Z are equal sets.

If X is not a proper subset of Y, it is denoted as $X \not\subset Y$.

Superset. If X is a subset of Y, then Y is called a *superset* of X. This is denoted by $Y \supseteq X$, i.e., $Y \supseteq X$ if and only if $X \subseteq Y$. If $X = \{1, 2, 3\}$ and $Y = \{1, 2, 3, 4, 5\}$, then $Y \supseteq X$.

Empty Set. A set with no elements is called an *empty set*. An empty set, denoted by \varnothing , is also referred to as a null or void set.

Power Set. The set of all subsets of a set X is called the *power set* of X. It is represented as $\wp(X)$. If $X = \{a, b, c\}$, then $\wp(X) = \{\varnothing, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$. If $|X| = n$, then $|\wp(X)| = 2^n$.

Venn Diagrams. Venn diagrams are graphic representations of sets as enclosed areas in the plane. In Figure TBD.2, the rectangle represents the universal set and the shaded region represents a set X.

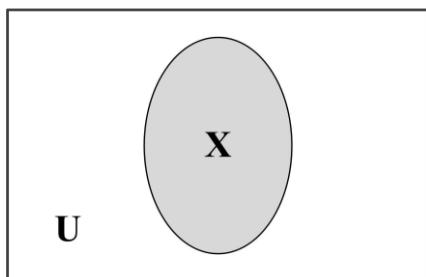


Figure TBD.2. Venn Diagram for set X

2.1. Set Operations

Intersection. The intersection of two sets X and Y, denoted by $X \cap Y$, is the set of

common elements in both X and Y. In other words, $X \cap Y = \{p \mid (p \in X) \wedge (p \in Y)\}$. For example, $\{1, 2, 3\} \cap \{3, 4, 6\} = \{3\}$

If $X \cap Y = \varnothing$, then the two sets X and Y are said to be disjoint.

A Venn diagram for set intersection is shown in Figure tbd.3. The common portion of the two sets represents the set intersection.

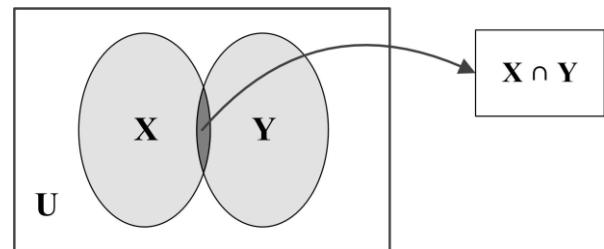


Figure tbd.3. Intersection of sets X and Y

Union. The union of two sets X and Y, denoted by $X \cup Y$, is the set of all elements either in X, or in Y, or in both. In other words, $X \cup Y = \{p \mid (p \in X) \vee (p \in Y)\}$. For example, $\{1, 2, 3\} \cup \{3, 4, 6\} = \{1, 2, 3, 4, 6\}$

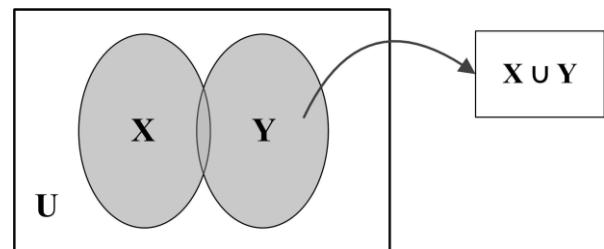


Figure tbd.4. Union of sets X and Y

It may be noted that $|X \cup Y| = |X| + |Y| - |X \cap Y|$.

A Venn diagram illustrating the union of two sets is represented by the shaded region in Figure tbd.4.

Complement. The set of elements in the universal set that do not belong to a given set X is called its complement set X' . In other words, $X' = \{p \mid (p \in U) \wedge (p \notin X)\}$.

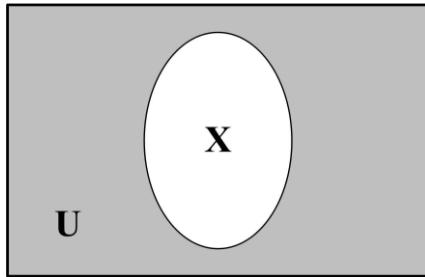


Figure tbd.5. Venn diagram for complement set of X

The shaded portion of the Venn diagram in Figure tbd.5 represents the complement set of X.

Set Difference or Relative Complement. The set of elements that belong to set X but not to set Y builds the set difference of Y from X. This is represented by $X - Y$. In other words, $X - Y = \{p \mid (p \in X) \wedge (p \notin Y)\}$. For example, $\{1, 2, 3\} - \{3, 4, 6\} = \{1, 2\}$.

It may be proved that $X - Y = X \cap Y'$. Set difference $X - Y$ is illustrated by the shaded region in Figure tbd.6 using a Venn diagram.

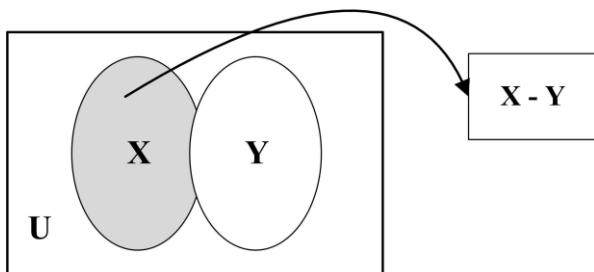


Figure tbd.6. Venn diagram for $X - Y$

Cartesian Product. An ordinary pair $\{p, q\}$ is a set with two elements. In a set, the order of the elements is irrelevant, so $\{p, q\} = \{q, p\}$.

In an ordered pair (p, q) , the order of occurrences of the elements is relevant. Thus, $(p, q) \neq (q, p)$ unless $p = q$. In general $(p, q) = (s, t)$ if and only if $p = s$ and $q = t$.

Given two sets X and Y, their Cartesian product $X \times Y$ is the set of all ordered pairs (p, q) such that $p \in X$ and $q \in Y$. In other words, $X \times Y = \{(p, q) \mid (p \in X) \wedge (q \in Y)\}$. For example, $\{a, b\} \times \{1, 2\} = \{(a, 1), (a, 2), (b, 1), (b, 2)\}$

2.2. Properties of Set

Some of the important properties and laws of sets are mentioned below.

1. Associative Laws:

$$X \cup (Y \cup Z) = (X \cup Y) \cup Z$$

$$X \cap (Y \cap Z) = (X \cap Y) \cap Z$$

2. Commutative Laws:

$$\begin{aligned} X \cup Y &= Y \cup X & X \cap Y \\ &= Y \cap X \end{aligned}$$

3. Distributive Laws:

$$X \cup (Y \cap Z) = (X \cup Y) \cap (X \cup Z)$$

$$X \cap (Y \cup Z) = (X \cap Y) \cup (X \cap Z)$$

4. Identity Laws:

$$\begin{aligned} X \cup \varnothing &= X & X \cap U \\ &= X \end{aligned}$$

5. Complement Laws:

$$\begin{aligned} X \cup X' &= U & X \cap X' \\ &= \varnothing \end{aligned}$$

6. Idempotent Laws:

$$\begin{aligned} X \cup X &= X & X \cap X \\ &= X \end{aligned}$$

7. Bound Laws:

$$\begin{aligned} X \cup U &= U & X \cap \varnothing \\ &= \varnothing \end{aligned}$$

8. Absorption Laws:

$$\begin{aligned} X \cup (X \cap Y) &= X & X \cap (X \cup Y) = \\ &= X \end{aligned}$$

9. De Morgan's Laws:

$$\begin{aligned} (X \cup Y)' &= X' \cap Y' & (X \cap Y)' = X' \\ &\cup Y' \end{aligned}$$

2.3. Relation and Function

A relation is an association between two sets

of information. Consider a set of residents of a city and their phone numbers. The pairing of names with corresponding phone numbers is a relation. This pairing is *ordered* for the entire relation. For each pair, either the name comes first followed by the phone number or the reverse. The set from which the first element is drawn is called the *domain set* and the other set is called the *range set*. The domain is what you start with, and the range is what you end with.

A function is a *well-behaved* relation. A relation $R(X, Y)$ is well behaved if the function maps every element of the domain set X to a single element of the range set Y . Consider domain set X as a set of persons and range set Y as their phone numbers. If a person may have more than one phone number, then this relation is not a function. However, if we draw a relation between names of residents and their date of births with the name set as domain, then this becomes a well-behaved relation and hence a function. This means that, while all functions are relations, not all relations are functions. In case of a function given an x , one gets one and exactly one y for each ordered pair (x, y) .

For example, consider the following two relations.

$$A: \{(3, -9), (5, 8), (7, -6), (3, 9), (6, 3)\}.$$

$$B: \{(5, 8), (7, 8), (3, 8), (6, 8)\}.$$

Are these functions as well?

In relation A, the domain is all x -values, i.e., $\{3, 5, 6, 7\}$, and the range is all y -values, i.e., $\{-9, -6, 3, 8, 9\}$.

Relation A is not a function, as there are two different range values, -9 and 9 , for the same x -value, 3 .

In relation B, the domain is same as for A, i.e., $\{3, 5, 6, 7\}$. However, the range is a

single element $\{8\}$. This qualifies as a function even if all x -values are mapped to the same y -value. Here, each x -value is distinct and hence the function is well behaved. Relation B may be represented by the equation $y = 8$.

The characteristic of a function may be verified using the vertical line test, which is stated below:

Given the graph of a relation, if one can draw a vertical line that crosses the graph in more than one place, then that relation is not a function.

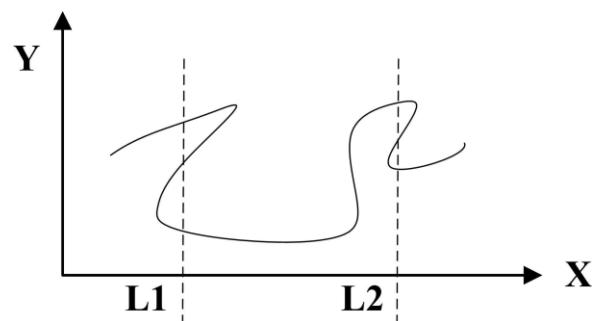


Figure tbd.7. Vertical line test for function

In Figure tbd.7, both lines L_1 and L_2 cut the graph for the relation thrice. This signifies that for the same x -value, there are three different y -values for each. Thus, the relation is not a function.

4. Graph and Tree

[1*, c10, c11]

4.1. Graph

A graph $G = (V, E)$ where V is the set of vertices (nodes) and E is the set of edges. Edges are also referred to as arcs or links.

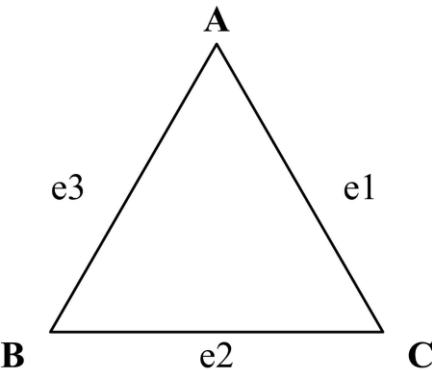


Figure tbd.8. Example of a graph

F is a function that maps the set of edges E to a set of ordered or unordered pairs of elements V . In Figure tbd.8, $G = (V, E)$ where $V = \{A, B, C\}$, $E = \{e1, e2, e3\}$, and $F = \{(e1, (A, C)), (e2, (C, B)), (e3, (B, A))\}$.

The graph in Figure tbd.8 is a simple graph that consists of a set of vertices or nodes and a set of edges connecting unordered pairs. The edges in simple graphs are undirected. Such graphs are also referred to as undirected graphs. In Figure tbd.8, $(e1, (A, C))$ may be replaced by $(e1, (C, A))$ as the pair between vertices A and C is unordered. This holds for the other two edges as well.

In a multigraph, more than one edge may connect the same two vertices. Two or more connecting edges between the same pair of vertices may reflect multiple associations between the same two vertices. Such edges are called parallel or multiple edges. In Figure tbd.9, the edges $e3$ and $e4$ are both between A and B. Figure tbd.9 is a multigraph where edges $e3$ and $e4$ are multiple edges.

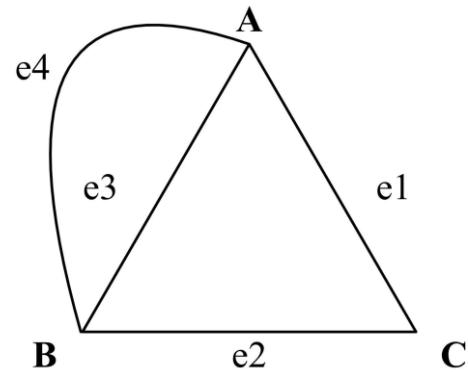


Figure tbd.9. Example of a multigraph

In a *pseudograph*, edges connecting a node to itself are allowed. Such edges are called loops.

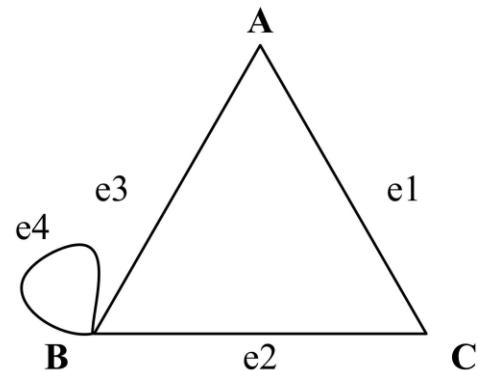


Figure tbd.10. Example of a pseudograph

In Figure tbd.10, the edge $e4$ both starts and ends at B. Figure tbd.10 is a pseudograph in which $e4$ is a loop.

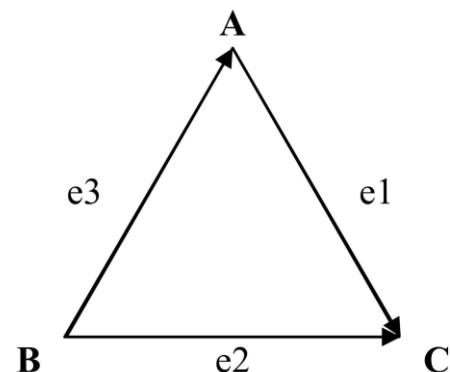


Figure tbd.11. Example of a directed graph

A directed graph $G = (V, E)$ consists of a set of vertices V and a set of edges E that are ordered pairs of elements of V . A directed graph may contain loops. In Figure tbd.11, $G = (V, E)$ where $V = \{A, B, C\}$, $E = \{e_1, e_2, e_3\}$, and $F = \{(e_1, (A, C)), (e_2, (B, C)), (e_3, (B, A))\}$.

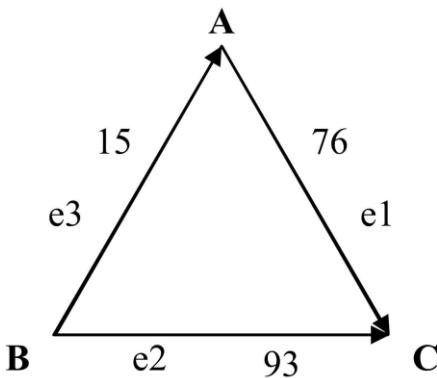


Figure tbd.12. Example of a weighted graph

In a weighted graph $G = (V, E)$, each edge has a weight associated with it. The weight of an edge typically represents the numeric value associated with the relationship between the corresponding two vertices. In Figure tbd.12, the weights for the edges e_1 , e_2 , and e_3 are taken to be 76, 93, and 15 respectively. If the vertices A, B, and C represent three cities in a state, the weights, for example, could be the distances in kilometers between these cities.

Let $G = (V, E)$ be an undirected graph with edge set E . Then, for an edge $e \in E$ where $e = \{u, v\}$, the following terminologies are often used:

- u, v are said to be *adjacent* or *neighbors* or *connected*.
 - edge e is *incident* with vertices u and v .
 - edge e *connects* u and v .
 - vertices u and v are *endpoints* for edge e .
- If vertex $v \in V$, the set of vertices in the

undirected graph $G(V, E)$, then:

- the *degree* of v , $\deg(v)$, is its number of incident edges, except that any self-loops are counted twice.
- a vertex with degree 0 is called an *isolated vertex*.
- a vertex of degree 1 is called a *pendant vertex*.

Let $G(V, E)$ be a directed graph. If $e(u, v)$ is an edge of G , then the following terminologies are often used:

- u is *adjacent to* v , and v is *adjacent from* u .
- e *comes from* u and *goes to* v .
- e *connects* u to v , or e *goes from* u to v .
- the *initial vertex* of e is u .
- the *terminal vertex* of e is v .

If vertex v is in the set of vertices for the directed graph $G(V, E)$, then

- *in-degree* of v , $\deg^-(v)$, is the number of edges going to v , i.e., for which v is the terminal vertex.
- *out-degree* of v , $\deg^+(v)$, is the number of edges coming from v , i.e., for which v is the initial vertex.
- *degree* of v , $\deg(v) = \deg^-(v) + \deg^+(v)$, is the sum of vs in-degree and out-degree.
- a loop at a vertex contributes 1 to both in-degree and out-degree of this vertex.

Following the definitions above, the degree of a node is unchanged whether we consider its edges to be directed or undirected.

In an undirected graph, a path of length n from u to v is a sequence of n adjacent edges from vertex u to vertex v .

- A path is a *circuit* if $u=v$.
- A path *traverses* the vertices along it.
- A path is *simple* if it contains no edge more than once.

A cycle on n vertices C_n for any $n \geq 3$ is a simple graph where $V = \{v_1, v_2, \dots, v_n\}$ and $E = \{\{v_1, v_2\}, \{v_2, v_3\}, \dots, \{v_{n-1}, v_n\}, \{v_n, v_1\}\}$.

For example, Figure tbd.13 illustrates two cycles of length 3 and 4.

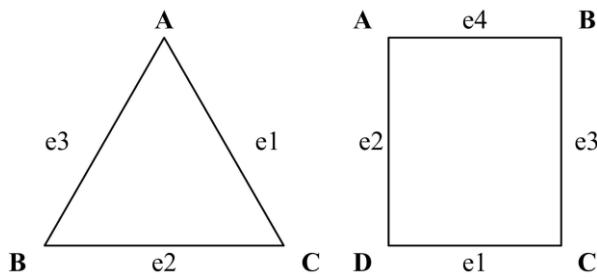


Figure tbd.13. Example of cycles C_3 and C_4

An adjacency list is a table with one row per vertex, listing its adjacent vertices. The adjacency listing for a directed graph maintains a listing of the terminal nodes for each of the vertex in the graph.

Vertex	Adjacency list
A	B, C
B	A, B, C
C	A, B

Figure tbd.14. Adjacency lists for graphs in Figures tbd.10 and tbd.11

Figure tbd.14 illustrates the adjacency lists for the pseudograph in Figure tbd.10 and the directed graph in Figure tbd.11. As the out-degree of vertex C in Figure tbd.11 is zero, there is no entry against C in the adjacency list.

Different representations for a graph—like adjacency matrix, incidence matrix, and adjacency lists—need to be studied.

4.2. Tree

A tree $T(N, E)$ is a hierarchical data structure of $n = |N|$ nodes with a specially designated root node R while the remaining $n - 1$ nodes

form subtrees under the root node R. The number of edges $|E|$ in a tree would always be equal to $|N| - 1$.

The subtree at node X is the subgraph of the tree consisting of node X and its descendants and all edges incident to those descendants. As an alternate to this recursive definition, a tree may be defined as a connected undirected graph with no simple circuits.

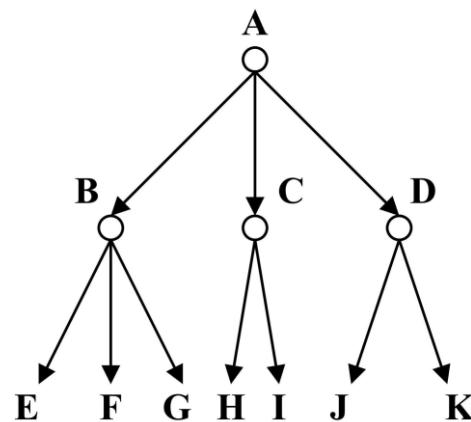


Figure tbd.15. Example of a tree

However, one should remember that a tree is strictly hierarchical in nature as compared to a graph, which is flat. In a tree, an ordered pair is built between two nodes as parent and child. Each child node in a tree is associated with only one parent node, whereas this restriction becomes meaningless for a graph where no parent-child association exists.

An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.

Figure tbd.15 presents a tree $T(N, E)$ where the set of nodes $N = \{A, B, C, D, E, F, G, H, I, J, K\}$. The edge set E is $\{(A, B), (A, C), (A, D), (B, E), (B, F), (B, G), (C, H), (C, I), (D, J), (D, K)\}$.

The parent of a non-root node v is the unique node u with a directed edge from u to v . Each node in the tree has a unique parent node

except the root of the tree. In Figure tbd.15, root node A is the parent node for nodes B, C, and D. Similarly, B is the parent of E, F, G, and so on. The root node A does not have any parent.

A node that has children is called an internal node. In Figure tbd.15, node A and node B are examples of internal nodes.

The degree of a node in a tree is the same as its number of children. In Figure tbd.15, root node A and its child B are both of degree 3. Nodes C and D have degree 2.

The distance of a node from the root node in terms of number of hops is called its *level*. Nodes in a tree are at different levels. The root node is at level 0. Alternately, the level of a node X is the length of the unique path from the root of the tree to node X. Root node A is at level 0 in Figure tbd.15. Nodes B, C, and D are at level 1. The remaining nodes in Figure tbd.15 are at level 2.

The height of a tree is the maximum of the levels of nodes in the tree. In Figure tbd.15, the height of the tree is 2.

A node is called a *leaf* if it has no children. The degree of a leaf node is 0. In Figure tbd.15, nodes E through K are leaf nodes with degree 0.

The ancestors or predecessors of a non-root node X are all the nodes in the path from root to node X. In Figure tbd.15, nodes A and D form the set of ancestors for J.

The successors or descendants of a node X are all the nodes that have X as its ancestor. For a tree with n nodes, all remaining $n-1$ nodes are successors of the root node. In Figure tbd.15, node B has successors in E, F, and G.

If node X is an ancestor of node Y, then node Y is a successor of X.

Two or more nodes sharing the same parent node are called *sibling* nodes. In Figure 15, nodes E and G are siblings. However, nodes

E and J, though from the same level, are not sibling nodes.

Two sibling nodes are of the same level, but two nodes in the same level are not necessarily siblings.

A tree is called an *ordered tree* if the relative position of occurrences of children nodes is significant. For example, a family tree is an ordered tree if, as a rule, the name of an elder sibling always appears before (i.e., on the left of) the younger sibling.

In an unordered tree, the relative position of occurrences between the siblings does not bear any significance and may be altered arbitrarily.

A binary tree is formed with zero or more nodes where there is a root node R and all the remaining nodes form a pair of ordered subtrees under the root node. In a binary tree, no internal node can have more than 2 children. However, one must consider that besides this criterion in terms of the degree of internal nodes, a binary tree is always ordered. If the positions of the left and right subtrees for any node in the tree are swapped, then a new tree is derived.

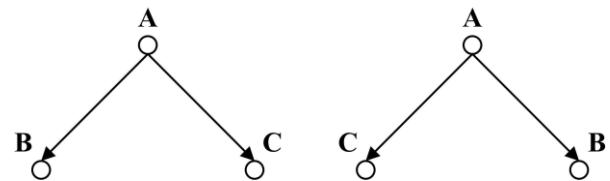


Figure tbd.16. Examples of binary trees

In Figure tbd.16, the two binary trees are different as the positions of occurrences of the children of A are different in the two trees.

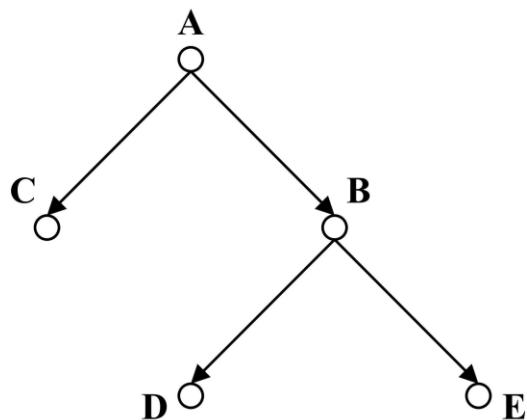


Figure tbd.17. Example of a full binary tree

According to [1*], a binary tree is called a full binary tree if every internal node has exactly 2 children. The binary tree in Figure tbd.17 is a full binary tree as both internal nodes A and B are of degree 2.

A full binary tree following the definition above is also referred to as a *strictly binary tree*.

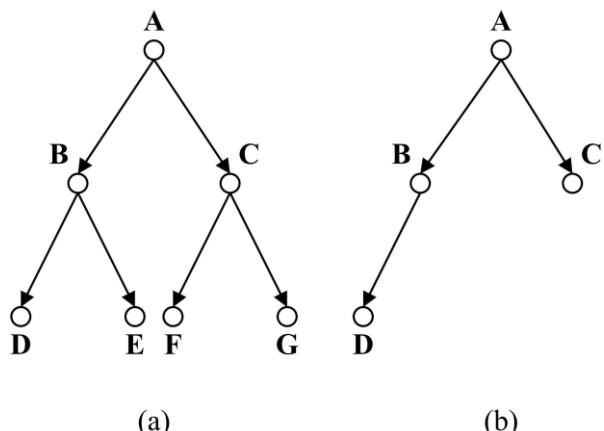


Figure tbd.18. Example of complete binary trees

Both binary trees in Figure tbd.18 are complete binary trees. The tree in Figure tbd.18(a) is a complete as well as a full binary tree. A complete binary tree has all its levels,

except possibly the last one, filled. In case the last level of a complete binary tree is not full, nodes occur from the leftmost positions available.

Interestingly, following the definitions above, the tree in Figure tbd.18(b) is a complete but not full binary tree as node B has only one child in D. On the contrary, the tree in Figure tbd.17 is a full but not complete binary tree as the children of B occur in the tree while the children of C do not appear in the last level.

A binary tree of height H is balanced if all leaf nodes occur at levels H or H – 1. All three binary trees in Figures tbd.17 and tbd.18 are balanced binary trees.

There are at most 2^H leaves in a binary tree of height H. In other words, if a binary tree with L leaves is full and balanced, then its height is $H = \lceil \log_2 L \rceil$. This is true for the two trees in Figures tbd.17 and tbd.18(a) as both trees are full and balanced. However, the tree in Figure tbd.18(b) is not a full binary tree.

A binary search tree (BST) is a special kind of binary tree in which each node contains a distinct key value, and the key value of each node in the tree is less than every key value in its right subtree and greater than every key value in its left subtree.

A *traversal algorithm* is a procedure for systematically visiting every node of a binary tree. Tree traversals may be defined recursively.

If T is binary tree with root R and the remaining nodes form an ordered pair of nonnull left subtree T_L and nonnull right subtree T_R below R, then the preorder traversal function PreOrder(T) is defined as:

$$\text{PreOrder}(T) = R, \quad \text{PreOrder}(T_L), \\ \text{PreOrder}(T_R) \dots \text{eqn. 1}$$

The recursive process of finding the preorder traversal of the subtrees continues until the subtrees are found to be Null. Here, commas

have been used as delimiters for the sake of improved readability.

The postorder and in-order may be similarly defined using eqn. 2 and eqn. 3 respectively.

$$\text{PostOrder}(T) = \text{PostOrder}(T_L), \text{PostOrder}(T_R), R \dots \text{eqn } 2$$

$$\text{InOrder}(T) = \text{InOrder}(T_L), R, \text{InOrder}(T_R) \dots \text{eqn } 3$$

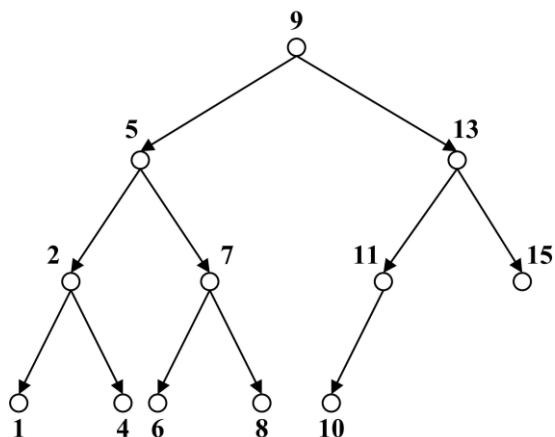


Figure tbd.19. A binary search tree

The tree in Figure tbd.19 is a binary search tree (BST). The preorder, postorder, and in-order traversal outputs for this BST are given below in their respective order.

Preorder output: 9, 5, 2, 1, 4, 7, 6, 8, 13, 11, 10, 15

Postorder output: 1, 4, 2, 6, 8, 7, 5, 10, 11, 15, 13, 9

In-order output: 1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 13, 15

Further discussion on trees and their usage has been included in section 6, Data Structure and Representation, of the Computing Foundations KA. <<<TBD!!! Double check!!!>>>

5. Finite State Machine

[1*, c13]

A computer system may be abstracted as a mapping from state to state driven by inputs. In other words, a system may be considered as a transition function $T: S \times I \rightarrow S \times O$, where S is the set of states and I, O are the input and output functions.

If the state set S is finite, the system is called a *finite state machine* (FSM).

Alternately, a finite state machine (FSM) is a mathematical abstraction composed of a finite number of states and transitions between those states. If the domain $S \times I$ is reasonably small, then one can specify T explicitly using diagrams similar to a flow graph to illustrate the way logic flows for different inputs. However, this is practical only for machines that have a very small information capacity.

An FSM has a finite internal memory, an input feature that reads symbols one at a time in a sequence, and an output feature.

The operation of an FSM begins from a start state, goes through transitions depending on input to different states, and can end in any valid state. However, only a few of all the states mark a successful flow of operation. These are called *accept states*.

The information capacity of an FSM is $C = \log |S|$. Thus, if we represent a machine having an information capacity of C bits as an FSM, then its state transition graph will have $|S| = 2^C$ nodes.

A finite state machine is formally defined as $M = (S, I, O, f, g, s_0)$.

S is the state set;

I is the set of input symbols;

O is the set of output symbols;

f is the state transition function;

g is the output function;

and s_0 is the initial state.

Given an input $x \in I$ on state S_k , the FSM makes a transition to state S_h following state transition function f and produces an output y

$\in O$ using the output function g .

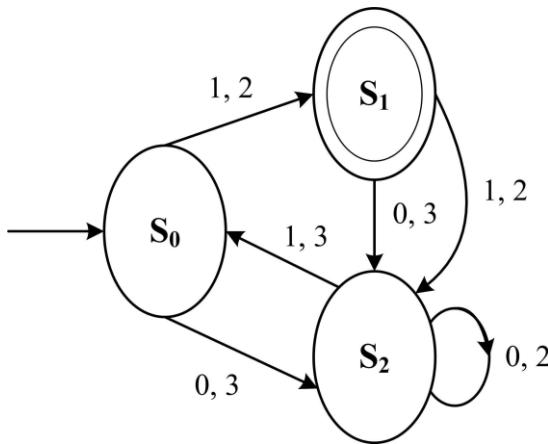


Figure tbd.20. Example of an FSM

Figure tbd.20 illustrates an FSM with S_0 as the start state and S_1 as the final state. Here, $S = \{S_0, S_1, S_2\}$; $I = \{0, 1\}$; $O = \{2, 3\}$; $f(S_0, 0) = S_2$, $f(S_0, 1) = S_1$, $f(S_1, 0) = S_2$, $f(S_1, 1) = S_2$, $f(S_2, 0) = S_2$, $f(S_2, 1) = S_0$; $g(S_0, 0) = 3$, $g(S_0, 1) = 2$, $g(S_1, 0) = 3$, $g(S_1, 1) = 2$, $g(S_2, 0) = 2$, $g(S_2, 1) = 3$.

Current State	Input		State Trans g
	0	1	
S_0	$S_2, 3$	$S_1, 2$	
S_1	$S_2, 3$	$S_2, 2$	
S_2	$S_2, 2$	$S_0, 3$	

Current State	Output f		State Trans g
	Input	Input	
	0	1	
S_0	3	2	S_2, S_1
S_1	3	2	S_2, S_2
S_2	2	3	S_2, S_0

(b)

(a)

Figure tbd.21. Tabular representation of an FSM

The state transition and output values for different inputs on different states may instead be represented using a state table. The state table for the FSM in Figure tbd.20 is shown in Figure tbd.21. Each pair against an

input symbol represents the new state and the output symbol. Figures tbd.21(a) and tbd.21(b) are alternate representations of the FSM in Figure tbd.20.

6. Grammar

[1*, c13]

The grammar of a natural language defines whether a combination of words makes a valid sentence. Unlike natural languages, a formal language is specified by a well-defined set of rules for syntaxes. The valid sentences of a formal language can be described by a grammar with the help of these rules, referred to as *production rules*.

A formal language is a set of finite-length words or strings over some finite alphabet, and a grammar specifies the rules for formation of those words or strings. The entire set of words that are valid for a grammar constitutes the language for the grammar. Thus, the grammar G is any compact, precise mathematical definition of a language L as opposed to just a raw listing of all legal sentences or examples of those sentences in that language.

A grammar implies an algorithm that would generate all legal sentences of the language. There are different types of grammars.

A phrase-structure or Type-0 grammar $G = (V, T, S, P)$ is a 4-tuple in which:

- V is the vocabulary, i.e., set of words.
- $T \subseteq V$ is a set of words called terminals.
- $S \in N$ is a special word called the start symbol.
- P is the set of production rules for substituting one sentence fragment for another.

There exists another set $N = V - T$ of words called nonterminals. The nonterminals represent concepts like *noun*. Production rules are applied on strings containing nonterminals until no more nonterminal symbols are present in the string. The start

symbol S is a nonterminal.

The language generated by a formal grammar G, denoted by L(G), is the set of all strings over the set of alphabets V that can be generated, starting with the start symbol, by applying production rules until all the nonterminal symbols are replaced in the string.

For example, let $G = (\{S, A, a, b\}, \{a, b\}, S, \{S \rightarrow aA, S \rightarrow b, A \rightarrow aa\})$. Here, the set of terminals are $N = \{S, A\}$, where S is the start symbol. The three production rules for the grammar are given as P1: $S \rightarrow aA$; P2: $S \rightarrow b$; P3: $A \rightarrow aa$.

Applying the production rules in all possible ways, the following words may be generated from the start symbol.

$$\begin{array}{ll} S \rightarrow aA & \text{(using P1 on} \\ \text{start symbol)} \\ \quad \rightarrow aaa & \text{(using P3)} \\ S \rightarrow b & \text{(using P2 on} \\ \text{start symbol)} \end{array}$$

Nothing else can be derived for G. Thus, the language of the grammar G consists of only two words: $L(G) = \{aaa, b\}$.

8.1 Language Recognition

Formal grammars can be classified according to the types of productions that are allowed. The Chomsky hierarchy (introduced by Noam Chomsky in 1956) describes such a classification scheme.

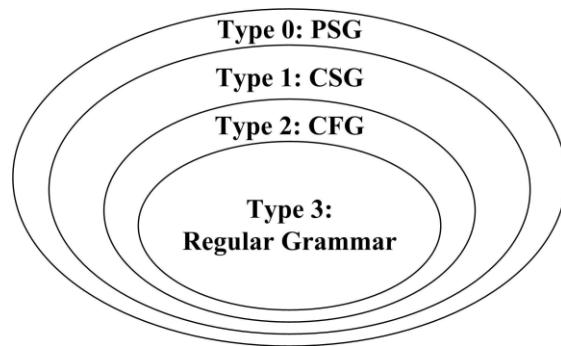


Figure tbd.22. Chomsky hierarchy of

grammars

As illustrated in Figure tbd.22, we infer the following on different types of grammars:

1. Every regular grammar is a context-free grammar (CFG).
2. Every CFG is a context-sensitive grammar (CSG).
3. Every CSG is a phrase-structure grammar (PSG).

Context-Sensitive Grammar: All fragments in the RHS are either longer than the corresponding fragments in the LHS or empty, i.e., if $b \rightarrow a$, then $|b| < |a|$ or $a = \varphi$. A formal language is context-sensitive if a context-sensitive grammar generates it.

Context-Free Grammar: All fragments in the RHS are of length 1, i.e., if $A \rightarrow a$, then $|A| = 1$ for all $A \in N$. The term context-free derives from the fact that A can always be replaced by a, regardless of the context in which it occurs. A formal language is context-free if a context-free grammar generates it. Context-free languages are the theoretical basis for the syntax of most programming languages.

Regular Grammar. All fragments in the RHS are either single terminals or a pair built by a terminal and a nonterminal; i.e., if $A \rightarrow a$, then either $a \in T$, or $a = cD$, or $a = Dc$ for $c \in T, D \in N$.

If $a = cD$, then the grammar is called a right linear grammar. On the other hand, if $a = Dc$, then the grammar is called a left linear grammar. Both the right linear and left linear grammars are regular or Type-3 grammar.

The language $L(G)$ generated by a regular grammar G is called a regular language.

A regular expression A is a string (or pattern) formed from the following six pieces of information: $a \in \Sigma$, the set of alphabets, ϵ , 0 and the operations, OR (+), PRODUCT (.), CONCATENATION (*). The language of G,

$L(G)$ is equal to all those strings that match G , $L(G) = \{x \in \Sigma^* | x \text{ matches } G\}$.

For any $a \in \Sigma$, $L(a) = a$; $L(\epsilon) = \{\epsilon\}$; $L(0) = 0$.
+ functions as an or, $L(A + B) = L(A) \cup L(B)$.
. creates a product structure, $L(AB) = L(A) \cdot L(B)$.

* denotes concatenation, $L(A^*) = \{x_1x_2\dots x_n | x_i \in L(A) \text{ and } n \geq 0\}$

For example, the regular expression $(ab)^*$ matches the set of strings: $\{\epsilon, ab, abab, ababab, abababab, \dots\}$. The regular expression $(aa)^*$ matches the set of strings on one letter a that have even length. The regular expression $(aaa)^* + (aaaaa)^*$ matches the set of strings of length equal to a multiple of 3 or 5.

7. Number Theory

[1*, c4]

Number theory is one of the oldest branches of pure mathematics and one of the largest. Of course, it concerns questions about numbers, usually meaning whole numbers and fractional or rational numbers. The different types of numbers include integer, real number, natural number, complex number, rational number, etc.

1.1. Types of Numbers

Natural Numbers. This group of numbers starts at 1 and continues: 1, 2, 3, 4, 5, and so on. Zero is not in this group. There are no negative or fractional numbers in the group of natural numbers. The common mathematical symbol for the set of all natural numbers is N .

Whole Numbers. This group has all natural numbers plus the number 0.

Unfortunately, not everyone accepts the above definitions of natural and whole numbers. There seems to be no general agreement about whether to include 0 in the

set of natural numbers. Many mathematicians consider that, in Europe, the sequence of natural numbers traditionally started with 1 (0 was not even considered to be a number by the Greeks). In the 19th century, set theoreticians and other mathematicians started the convention of including 0 in the set of natural numbers.

Integers. This group has all the whole numbers in it and their negatives. The common mathematical symbol for the set of all integers is Z , i.e., $Z = \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$.

Rational Numbers. These are any numbers that can be expressed as a ratio of two integers. The common symbol for the set of all rational numbers is Q .

Rational numbers may be classified into three types, based on how the decimals act. The decimals either do not exist, e.g., 15 , or, when decimals do exist, they may terminate, as in 15.6 , or they may repeat with a pattern, as in $1.666\dots$, (which is $5/3$).

Irrational Numbers. These are numbers that cannot be expressed as an integer divided by an integer. These numbers have decimals that never terminate and never repeat with a pattern, e.g., π or $\sqrt{2}$.

Real Numbers. This group is made up of all the rational and irrational numbers. The numbers that are encountered when studying algebra are real numbers. The common mathematical symbol for the set of all real numbers is R .

Imaginary Numbers. These are all based on the imaginary number i . This imaginary number is equal to the square root of -1 . Any real number multiple of i is an imaginary number, e.g., $i, 5i, 3.2i, -2.6i$, etc.

Complex Numbers. A complex number is a combination of a real number and an imaginary number in the form $a + bi$. The real part is a , and b is called the imaginary part.

The common mathematical symbol for the set of all complex numbers is \mathbf{C} . For example, $2 + 3i$, $3 - 5i$, $7.3 + 0i$, and $0 + 5i$. $7.3 + 0i$ is the same as the real number 7.3 . All real numbers are complex numbers with zero for the imaginary part. Similarly, $0 + 5i$ is just the imaginary number $5i$. All imaginary numbers are complex numbers with zero for the real part.

1.1. Divisibility

Elementary number theory involves divisibility among integers. Let $a, b \in \mathbf{Z}$ with $a \neq 0$. The expression $a|b$, i.e., a divides b if $\exists c \in \mathbf{Z}: b = ac$, i.e., there is an integer c such that c times a equals b . For example, $3|12$ is true, but $3|7$ is false.

If a divides b , then we say that a is a factor of b or a is a divisor of b , and b is a multiple of a .

b is even if and only if $2|b$.

Let $a, d \in \mathbf{Z}$ with $d > 1$. Then $a \bmod d$ denotes that the remainder r from the division algorithm with dividend a and divisor d , i.e., the remainder when a is divided by d . We can compute $(a \bmod d)$ by: $a - d * \lfloor a/d \rfloor$, where $\lfloor a/d \rfloor$ represents the floor of the real number.

Let $\mathbf{Z}^+ = \{n \in \mathbf{Z} \mid n > 0\}$ and $a, b \in \mathbf{Z}$, $m \in \mathbf{Z}^+$, then a is congruent to b modulo m , written as $a \equiv b \pmod{m}$, if and only if $m | a - b$.

Alternately, a is congruent to b modulo m if and only if $(a - b) \bmod m = 0$.

1.1. Prime Number

An integer $p > 1$ is prime if and only if it is not the product of any two integers greater than 1, i.e., p is prime if $p > 1 \wedge \exists \neg a, b \in \mathbf{N}: a > 1, b > 1, a * b = p$.

The only positive factors of a prime p are 1 and p itself. The numbers 2, 13, 29, 61, etc. are prime numbers. Nonprime integers

greater than 1 are called composite numbers. A composite number may be composed by multiplying two integers greater than 1.

There are many interesting applications of prime numbers; among them are the public-key cryptography scheme, which involves the exchange of public keys containing the product $p * q$ of two random large primes p and q (a private key) that must be kept secret by a given party.

1.1. Greatest Common Divisor

The greatest common divisor $\gcd(a, b)$ of integers a, b is the greatest integer d that is a divisor both of a and of b , i.e.,

$$d = \gcd(a, b) \text{ for } \max(d: d|a \wedge d|b)$$

For example, $\gcd(24, 36) = 12$.

Integers a and b are called relatively prime or coprime if and only if their GCD is 1. Neither 35 nor 6 are prime, but they are coprime as these two numbers have no common factors greater than 1, so their GCD is 1.

A set of integers $X = \{i_1, i_2, \dots\}$ is relatively prime if all possible pairs i_h, i_k , $h \neq k$ drawn from the set X are relatively prime.

8. Basics of Counting

[1*, c6]

The sum rule states that if a task t_1 can be done in n_1 ways and a second task t_2 can be done in n_2 ways, and if these tasks cannot be done at the same time, then there are $n_1 + n_2$ ways to do either task.

- If A and B are disjoint sets, then $|A \cup B| = |A| + |B|$.
- In general if A_1, A_2, \dots, A_n are disjoint sets, then $|A_1 \cup A_2 \cup \dots \cup A_n| = |A_1| + |A_2| + \dots + |A_n|$.

If there are 200 athletes doing sprint events and 30 athletes who participate in the long jump event, then how many ways are there to

pick one athlete who is either a sprinter or a long jumper?

Using the sum rule, the answer would be $200 + 30 = 230$.

The product rule states that if a task t_1 can be done in n_1 ways and a second task t_2 can be done in n_2 ways after the first task has been done, then there are $n_1 * n_2$ ways to do the procedure.

- If A and B are disjoint sets, then $|A \times B| = |A| * |B|$.
- In general, if A_1, A_2, \dots, A_n are disjoint sets, then $|A_1 \times A_2 \times \dots \times A_n| = |A_1| * |A_2| * \dots * |A_n|$.

If there are 200 athletes doing sprint events and 30 athletes who participate in the long jump event, then how many ways are there to pick two athletes so that one is a sprinter and the other is a long jumper?

Using the product rule, the answer would be $200 * 30 = 6000$.

The *principle of inclusion-exclusion* states that if a task t_1 can be done in n_1 ways and a second task t_2 can be done in n_2 ways at the same time with t_1 , then to find the total number of ways the two tasks can be done, subtract the number of ways to do both tasks from $n_1 + n_2$.

- If A and B are not disjoint, $|A \cup B| = |A| + |B| - |A \cap B|$.

In other words, the principle of inclusion-exclusion aims to ensure that the objects in the intersection of two sets are not counted more than once.

Recursion is the general term for the practice of defining an object in terms of itself. There are recursive algorithms, recursively defined functions, relations, sets, etc.

A recursive function is a function that calls itself. For example, we can define $f(n) = 3 * f(n - 1)$ for all $n \in \mathbb{N}$ and $n \neq 0$ and $f(0) = 5$.

An algorithm is recursive if it solves a problem by reducing it to an instance of the

same problem with a smaller input.

A phenomenon is said to be *random* if individual outcomes are uncertain, but the long-term pattern of many individual outcomes is predictable.

The probability of any outcome for a random phenomenon is the proportion of times the outcome would occur in a very long series of repetitions.

The probability $P(A)$ of any event A satisfies $0 \leq P(A) \leq 1$. Any probability is a number between 0 and 1. If S is the sample space in a probability model, the $P(S) = 1$. All possible outcomes together must have probability of 1.

Two events A and B are disjoint if they have no outcomes in common and so can never occur together. If A and B are two disjoint events, $P(A \text{ or } B) = P(A) + P(B)$. This is known as the addition rule for disjoint events.

If two events have no outcomes in common, the probability that one or the other occurs is the sum of their individual probabilities.

Permutation is an arrangement of objects in which the order matters without repetition. One can choose r objects in a particular order from a total of n objects by using ${}^n P_r$ ways, where ${}^n P_r = n! / (n - r)!$. Various notations like ${}^n P_r$ and $P(n, r)$ are used to represent the number of permutations of a set of n objects taken r at a time.

Combination is a selection of objects in which the order does not matter without repetition. This is different from a permutation because the order does not matter. If the order is only changed (and not the members) then no new combination is formed. One can choose r objects in any order from a total of n objects by using ${}^n C_r$ ways, where ${}^n C_r = n! / [r! * (n - r)!]$.

9. Discrete Probability

[1*, c7]

Probability is the mathematical description of randomness. Basic definitions of probability and randomness have been provided in the previous section. Here, start with the concepts behind probability distribution and discrete probability.

A probability model is a mathematical description of a random phenomenon consisting of two parts: a sample space S and a way of assigning probabilities to events. The sample space defines the set of all possible outcomes, whereas an event is a subset of a sample space representing a possible outcome or a set of outcomes.

A random variable is a function or rule that assigns a number to each outcome. Basically, it is just a symbol that represents the outcome of an experiment. For example, let X be the number of heads when the experiment is flipping a coin n times. Similarly, let S be the speed of a car as measured on a radar detector.

The values for a random variable could be discrete or continuous depending on the experiment. A discrete random variable can hold all possible outcomes without missing any, although it might take an infinite amount of time. A continuous random variable is used to measure an uncountable number of values even if an infinite amount of time is given.

For example, if random variable X represents an outcome that is a real number between 1 and 100, then X may have an infinite number of values. One can never list all possible outcomes for X even if an infinite amount of time is allowed. Here, X is a continuous random variable. On the other hand, for the same interval of 1 to 100, another random variable Y can be used to list all integer values in the range. Here, Y is a discrete random variable.

An upper-case letter, say X, will represent the *name* of the random variable. Its lower-case counterpart, x, will represent the *value* of the

random variable.

The probability that the random variable X will equal x is:

$$P(X = x) \text{ or, more simply, } P(x).$$

A Probability Distribution (Density) Function (PDF) is a table, formula, or graph that describes the values of a random variable and the probability associated with these values. Probabilities associated with discrete random variables have the following properties:

- i. $0 \leq P(x) \leq 1$ for all x
- ii. $\sum P(x) = 1$

A discrete probability distribution can be represented as a discrete random variable.

X	1	2	3	4	5	6
P(x)	1/6	1/6	1/6	1/6	1/6	1/6

Figure tbd.23. A discrete probability function for a rolling die

The mean μ of a probability distribution model is the sum of the product terms for individual events and its outcome probability. In other words, for the possible outcomes x_1, x_2, \dots, x_n in a sample space S if p_k is the probability of outcome x_k , the mean of this probability would be $\mu = x_1 p_1 + x_2 p_2 + \dots + x_n p_n$. The mean of the probability density for the distribution in Figure tbd.23 would be

$$\begin{aligned} & 1 * (1/6) + 2 * (1/6) + 3 * (1/6) + 4 * (1/6) + \\ & 5 * (1/6) + 6 * (1/6) \\ & = 21 * (1/6) = 3.5 \end{aligned}$$

Here, the sample space refers to the set of all possible outcomes.

The variance σ^2 of a discrete probability model is: $\sigma^2 = (x_1 - \mu)^2 p_1 + (x_2 - \mu)^2 p_2 + \dots + (x_n - \mu)^2 p_n$. The *standard deviation*, σ , is the square root of the variance. For the

probability distribution in Figure tbd.23, the variation σ^2 would be

$$\begin{aligned}\sigma^2 &= [(1 - 3.5)^2 * (1/6) + (2 - 3.5)^2 * (1/6) + \\&\quad (3 - 3.5)^2 * (1/6) + (4 - 3.5)^2 * (1/6) + (5 - 3.5)^2 * (1/6) + (6 - 3.5)^2 * (1/6)] \\&= (6.25 + 2.25 + 0.25 + 0.5 + 2.25 + 6.25) * (1/6) \\&= 17.5 * (1/6) \\&= 2.90\end{aligned}$$

$$\therefore \text{standard deviation } \sigma = \sqrt{2.9} = 1.70$$

These numbers indeed aim to derive the average value from repeated experiments. This is based on the single most important phenomenon of probability, i.e., the average value from repeated experiments is likely to be close to the expected value of one experiment. Moreover, the average value is more likely to be closer to the expected value of any one experiment as the number of experiments increases.

10. Numerical Precision, Accuracy, and Error

[2*, c1]

The main goal of numerical analysis is to develop efficient algorithms for computing precise numerical values of functions, solutions of algebraic and differential equations, optimization problems, etc.

Digital computers can only store finite numbers. A digital computer cannot represent any infinitely large number—be it an integer, rational number, or any real or complex number [see section 7, Number Theory]. The mathematics of approximation becomes critical to handle numbers in the finite range a computer can handle.

Each number in a computer is assigned a location (e.g., an address or register), and consists of a quantity of binary digits or bits. A k bit location can store any one of $N = 2^k$

different numbers. A 32-bit location can store any one of $N = 2^{32} \approx 4.3 \times 10^9$ different numbers, while a 64-bit location can store any one of $N' = 2^{64} \approx 1.84 \times 10^{19}$ different numbers. The question is how to distribute those numbers for maximum efficiency and accuracy in practical computations.

One choice is to distribute the numbers evenly, leading to fixed-point arithmetic. In this system, the first bit represents the sign and the remaining bits represent magnitude. The decimal point—more appropriately, the binary point: the transition between whole and fractional values—can be anywhere. Integer numbers are represented by placing the binary point immediately to the right of the least significant bit, and integer numbers between $-2^{k-1}-1$ and 2^{k-1} can be stored. Placing the binary point to the left of the least-significant bit allows non-integer values to be represented.

Another choice is to space the numbers closely together—say with a uniform gap of 2^{-n} —and so distribute the total N numbers uniformly over the interval $-2^{-n-1}N < x \leq 2^{-n-1}N$. Real numbers lying between the gaps are represented by either *rounding* (meaning the closest exact representative) or *chopping* (meaning the exact representative immediately below—or above, if negative—the number).

Numbers outside the range must be represented by the largest (or largest negative) number that can be represented. This becomes a symbol for overflow, which occurs when a computation produces a value outside of the range.

When processing speed is a significant bottleneck, fixed-point representations can be an attractive and faster alternative to the more cumbersome floating-point arithmetic most commonly used in practice.

Accuracy and *precision* are very important terms in numerical analysis.

Accuracy is the closeness with which a measured or computed value agrees with the true value.

Precision, on the other hand, is the closeness with which two or more measured or computed values for the same thing agree with each other. In other words, precision is the closeness with which a number represents an exact value.

Let x be a real number and let x^* be an approximation. The *absolute error* in the approximation $x^* \approx x$ is defined as $|x^* - x|$. The *relative error* is defined as the ratio of the absolute error to the size of x , i.e., $|x^* - x| / |x|$, which assumes $x \neq 0$; otherwise, relative error is not defined. For example, 1000000 is an approximation of 1000001 with an absolute error of 1 and a relative error of 10^{-6} , while 10 is an approximation of 11 with an absolute error of 1 and a relative error of 0.1. Typically, relative error is more intuitive and the preferred determiner of the size of the error. The present convention is that errors are always ≥ 0 and are = 0 if and only if the approximation is exact.

An approximation x^* has k significant decimal digits if its relative error is $< 5 \times 10^{-k-1}$. This means that the first k digits of x^* following its first nonzero digit are the same as those of x .

Significant digits are the digits of a number that are known to be correct. In a measurement, one uncertain digit is included. For example, measurement of length with a ruler of 15.5 mm with ± 0.5 mm maximum allowable error has 2 significant digits, whereas a measurement of the same length using a caliper and recorded as 15.47 mm with ± 0.01 mm maximum allowable error has 3 significant digits.

11. Algebraic Structures

This section introduces a few representations used in higher algebra. An algebraic structure

consists of one or two sets closed under some operations and satisfying several axioms, including none. For example, group, monoid, ring, and lattice are examples of algebraic structures. Group, monoid, and ring are defined in this section.

11.1 Group

A set S closed under a binary operation \bullet forms a group if the binary operation satisfies the following four criteria:

- Associative: $\forall a, b, c \in S$, the equation $(a \bullet b) \bullet c = a \bullet (b \bullet c)$ holds
- Identity: There exists an identity element $I \in S$ such that for all $a \in S$, $I \bullet a = a \bullet I = a$
- Inverse: Every element $a \in S$, has an inverse $a' \in S$ with respect to the binary operation, i.e., $a \bullet a' = I$; for example, the set of integers Z with respect to the addition operation is a group. The identity element of the set is 0 for the addition operation. $\forall x \in Z$, the inverse of x would be $-x$, which is also included in Z
- Closure property: $\forall a, b \in S$, the result of the operation $a \bullet b \in S$

A group that is commutative, i.e., $a \bullet b = b \bullet a$, is known as a commutative or Abelian group.

The set of natural numbers N (with the operation of addition) is not a group, since there is no inverse for any $x > 0$ in the set of natural numbers. Thus, the third criterion (of inverse) is violated. However, the set of natural number has some structure.

Sets with an associative operation (the first criterion) are called semigroups; if they also have an identity element (the second criterion), then they are called monoids. The set of natural numbers under addition is an example of a monoid, a structure that is not quite a group because it is missing the requirement that every element have an inverse under the operation.

A monoid is a set S that is closed under a

single associative binary operation \bullet and has an identity element $I \in S$ such that for all $a \in S$, $I \bullet a = a \bullet I = a$. A monoid must contain at least one element. The set of natural numbers N forms a commutative monoid under addition with identity element 0. The same set of natural numbers N also forms a monoid under multiplication with identity element 1. The set of positive integers P forms a commutative monoid under multiplication with identity element 1.

It may be noted that, unlike those in a group, elements of a monoid need not have inverses. A monoid can also be thought of as a semigroup with an identity element.

A *subgroup* is a group H contained within a bigger one, G , such that the identity element of G is contained in H , and whenever h_1 and h_2 are in H , then so are $h_1 \bullet h_2$ and h_1^{-1} . Thus, the elements of H , equipped with the group operation on G restricted to H , indeed form a group.

Given any subset S of a group G , the subgroup generated by S consists of products of elements of S and their inverses. It is the smallest subgroup of G containing S . For example, let G be the Abelian group whose elements are $G = \{0, 2, 4, 6, 1, 3, 5, 7\}$ and whose group operation is addition modulo 8. This group has a pair of nontrivial subgroups: $J = \{0, 4\}$ and $H = \{0, 2, 4, 6\}$, where J is also a subgroup of H .

In group theory, a cyclic group is a group that can be generated by a single element, in the sense that the group has an element a (called the *generator* of the group) such that, when written multiplicatively, every element of the group is a power of a .

A group G is cyclic if $G = \{a^n \text{ for any integer } n\}$.

Since any group generated by an element in a group is a subgroup of that group, showing that the only subgroup of a group G that contains a is G itself suffices to show that G

is cyclic. For example, the group $G = \{0, 2, 4, 6, 1, 3, 5, 7\}$, with respect to addition modulo 8 operation, is cyclic. The subgroups $J = \{0, 4\}$ and $H = \{0, 2, 4, 6\}$ are also cyclic.

11.2 Ring

If we take an Abelian group and define a second operation on it, a new structure is found that is different from just a group. If this second operation is associative and is distributive over the first, then we have a ring. A ring is a triple of the form $(S, +, \bullet)$, where $(S, +)$ is an Abelian group, (S, \bullet) is a semigroup, and \bullet is distributive over $+$; i.e., $\forall a, b, c \in S$, the equation $a \bullet (b + c) = (a \bullet b) + (a \bullet c)$ holds. Further, if \bullet is commutative, then the ring is said to be commutative. If there is an identity element for the \bullet operation, then the ring is said to have an identity.

As an example, $(Z, +, *)$, i.e., the set of integers Z , with the usual addition and multiplication operations, is a ring. As $(Z, *)$ is commutative, this ring is a commutative or Abelian ring. The ring has 1 as its identity element.

Note that the second operation may not have an identity element, nor do we need to find an inverse for every element with respect to this second operation. As for what distributive means, intuitively it is what we do in elementary mathematics when performing the following change: $a * (b + c) = (a * b) + (a * c)$.

A field is a ring for which the elements of the set, excluding 0, form an Abelian group with the second operation. A simple example of a field is the field of rational numbers $(R, +, *)$ with the usual addition and multiplication operations. The numbers of the format $a/b \in R$, where a, b are integers and $b \neq 0$. The additive inverse of such a fraction is simply $-a/b$, and the multiplicative inverse is b/a provided that $a \neq 0$.

12. Calculus

This branch of mathematics deals with derivatives and integrals of functions using methods originally based on summation of infinitesimal differences. The two elements of calculus are differential calculus and integral calculus.

- Differential calculus analyzes the rate of change of one quantity in relation to the rate of change of another. Geometrically, it is the slope of the tangent to the graph of the function. The rate of change of x with respect to y is expressed as dx/dy ;
- Integral calculus analyzes such concepts as the area or volume enclosed by a function.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	Rosen 2018 [1*]	Cheney and Kincaid 2020 [2*]
1. Basic Logic	c1	
2. Proof techniques	c1	
3. Set, Relation, Function	c2	
4. Graph and Tree	c10, c11	
5. Finite State Machine	c13	
6. Grammar	c13	
7. Number Theory	c4	
8. Basics of Counting	c6	
9. Discrete Probability	c7	
10. Numerical Precision, Accuracy, and Error		c2
11. Algebraic Structures		
12. Calculus		

REFERENCES

- [1*] K. Rosen, *Discrete Mathematics and its Applications*, 8th ed., McGraw-Hill, 2018.
- [2*] E.W. Cheney and D.R. Kincaid, *Numerical Mathematics and Computing*, 7th ed., Addison Wesley, 2020.

CHAPTER 18

ENGINEERING FOUNDATIONS

ACRONYMS

CAD	Computer-Aided Design
CMMI	Capability Maturity Model Integration
PDF	Probability Density Function
PMF	Probability Mass Function
RCA	Root Cause Analysis
SDLC	Software Development Life Cycle

INTRODUCTION

IEEE defines engineering as “the application of a systematic, disciplined, quantifiable approach to structures, machines, products, systems or processes” [1]. As the theory and

practice of software engineering matures, it is increasingly apparent that software engineering is an engineering discipline that is based on skills and knowledge common to all engineering disciplines. This Engineering Foundations knowledge area (KA) is concerned with the engineering foundations from other engineering disciplines that apply to software engineering. The focus is on topics that support other KAs while minimizing duplication of content covered elsewhere in this Guide.

BREAKDOWN OF TOPICS FOR ENGINEERING FOUNDATIONS

The breakdown of topics for the Engineering Foundations KA is shown in Figure TBD.1.

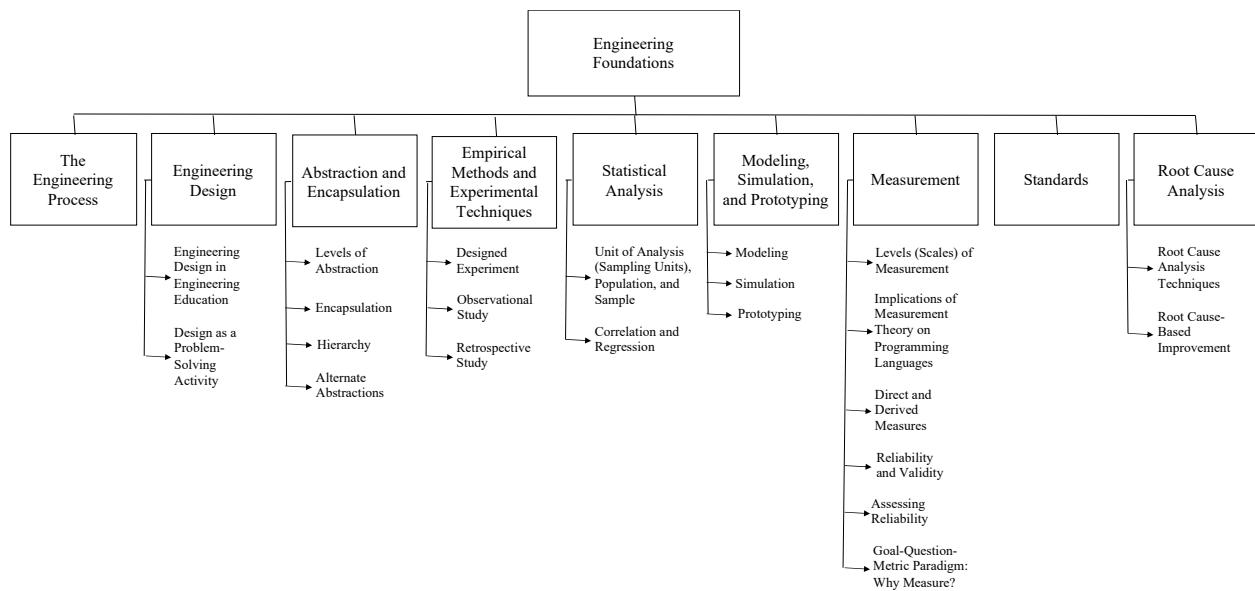


Figure TBD.1. Breakdown of Topics for the Engineering Foundations KA

1. The Engineering Process

[2*, ch4]

The engineering process, common to all engineering disciplines is discussed more fully in the Software Engineering Economics KA, refer to that chapter for a more complete discussion. A brief, high-level summary is included here. Figure TBD.2 shows the process flow.

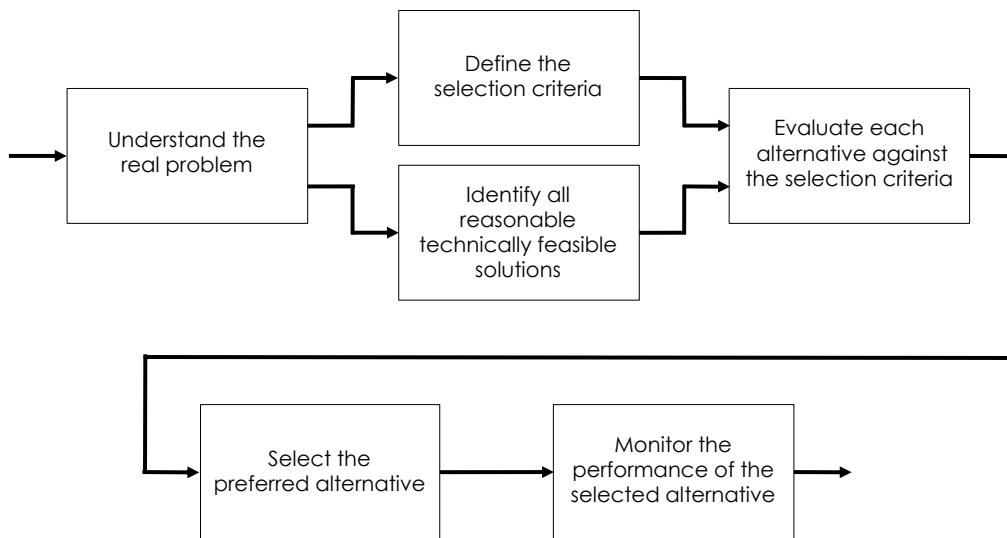


Figure TBD.2. The Engineering Process

The engineering process is necessarily iterative, knowledge gained at any point in the process may inform earlier steps and trigger iteration. These steps are briefly expanded below.

- Understand the real problem—engineering begins when a need is recognized, and no existing solution will meet that need. However, the problem that needs to be solved is not always the problem engineers are asked to solve. Use root cause techniques (later in this KA) to help discover what the real underlying problem needing solution is;
- Define the selection criteria—engineering decisions almost always involve financial criteria as discussed in

the Software Engineering Economics KA, but criteria beyond pure finance can also be relevant. Be sure that all relevant selection criteria have been identified;

- Identify all reasonable technically feasible solutions—the best solution is rarely the first solution that comes to mind. Be sure to consider multiple technically feasible solutions to help ensure that the optimal solution is in the set being considered;
- Evaluate each alternative against the selection criteria—determine how well each of the technically feasible solutions satisfies the need;
- Select the preferred alternative—identify which of the technically feasible solutions best satisfies the selection criteria;
- Monitor the performance of the selected alternative—the engineering process necessarily depends on estimates, and

those estimates might be wrong. Be sure to evaluate the real-world performance of the selected alternative and decide if, not too late, whether one of the other alternatives might be better.

Much of the remainder of this KA elaborates on details of this higher-level engineering process.

2. Engineering Design

[3*, c1s2-s4]

A product's life cycle costs are largely influenced by its design. This is true for manufactured products as well as for software. The design of software is guided by the features to be implemented and the quality attributes to be achieved. It is important to note that software engineers use "design" within their own context; while there are some commonalities, there are also many differences between engineering design as discussed in this section and software engineering design as discussed in the Software Architecture KA and Software Design KA. The scope of engineering design is generally viewed as much broader than that of software design.

Many disciplines engage in problem solving activities where there is a single correct solution. In engineering, most problems have many solutions and the focus is on finding a feasible solution (among many alternatives) that best meets the needs presented. The set of possible solutions is often constrained by explicitly imposed limitations such as cost, available resources, and the state of discipline or domain knowledge. In engineering problems, sometimes there are also implicit constraints (such as the physical properties of materials or laws of physics) that also restrict the set of feasible solutions for a given problem.

2.1. Engineering Design in Engineering Education

The importance of engineering design in engineering education can be clearly seen by the high expectations held by various accreditation bodies for engineering education. Both the Canadian Engineering Accreditation Board and the Accreditation Board for Engineering and Technology (ABET) note the importance of including engineering design in education programs.

The Canadian Engineering Accreditation Board includes requirements for the amount of engineering design experience/coursework that is necessary for engineering students as well as qualifications for the faculty members who teach such coursework or supervise design projects. Their accreditation criteria states:

Design: An ability to design solutions for complex, open-ended engineering problems and to design systems, components or processes that meet specified needs with appropriate attention to health and safety risks, applicable standards, and economic, environmental, cultural, and societal considerations. [4, p7]

In a similar manner, ABET defines engineering design as

a process of devising a system, component, or process to meet desired needs and specifications within constraints. It is an iterative, creative, decision-making process in which the basic sciences, mathematics, and engineering sciences are applied to convert resources into solutions. [5, p7]

Thus, it is clear that engineering design is a vital component in the training and education for all engineers. The remainder of this section will focus on various aspects of engineering design.

2.2. Design as a Problem-Solving Activity

[3*, c1s4, c2s1, c3s3] [6*, c5s1]

Engineering design is primarily a problem-solving activity. Design problems tend to be open ended and more vaguely defined. There are usually several alternative ways to solve the same problem. Design is generally considered to be a *wicked problem*—a term first coined by Horst Rittel in the 1960s when design methods were a subject of intense interest. Rittel sought an alternative to the linear, step-by-step model of the design process being explored by many designers and design theorists and argued that most of the problems addressed by the designers are wicked problems. As explained by Steve McConnell, a wicked problem is one that could be clearly defined only by solving it or by solving part of it. This paradox implies, essentially, that a wicked problem has to be solved once in order to define it clearly and then solved again to create a solution that works. This has been an important insight for software designers for several decades [6*, c5s1].

3. Abstraction and Encapsulation

[6*, c5s2–4]

Abstraction is an indispensable technique associated with problem solving. It refers to both the process and result of generalization by reducing the information of a concept, a problem, or an observable phenomenon so that one can focus on the “big picture.” One of the most important skills in any engineering undertaking is framing the levels of abstraction appropriately.

“Through abstraction,” according to Voland, “we view the problem and its possible solution paths from a higher level of conceptual understanding. As a result, we may become better prepared to recognize possible relationships between different

aspects of the problem and thereby generate more creative design solutions” [2*]. This is particularly true in computer science in general (such as hardware vs. software) and in software engineering in particular (data structure vs. data flow, and so forth).

According to Dijkstra, “The purpose of abstracting is not to be vague, but to create a new semantic level in which one can be absolutely precise” [7].

1.1. Levels of Abstraction

When abstracting, we concentrate on one “level” of the big picture at a time with confidence that we can then connect effectively with levels above and below. Although we focus on one level, abstraction does not mean knowing nothing about the neighboring levels. Abstraction levels do not necessarily correspond to discrete components in reality or in the problem domain, but to well-defined standard interfaces such as programming APIs. The advantages that standard interfaces provide include portability, easier software/hardware integration and wider usage.

1.1. Encapsulation

Encapsulation is a mechanism used to implement abstraction. When we are dealing with one level of abstraction, the information concerning the levels below and above that level is encapsulated. This information can be the concept, problem, or observable phenomenon; or it may be the permissible operations on these relevant entities. Encapsulation usually comes with some degree of information hiding in which some or all underlying details are hidden from the level above the interface provided by the abstraction. To an object, information hiding means we don’t need to know the details of how the object is represented or how the operations on those objects are implemented.

1.1. Hierarchy

When we use abstraction in our problem formulation and solution, we may use different abstractions at different times—in other words, we work on different levels of abstraction as the situation calls. Most of the time, these different levels of abstraction are organized in a hierarchy. There are many ways to structure a particular hierarchy and the criteria used in determining the specific content of each layer in the hierarchy varies depending on the individuals performing the work.

Sometimes, a hierarchy of abstraction is sequential, which means that each layer has one and only one predecessor (lower) layer and one and only one successor (upper) layer—except the upmost layer (which has no successor) and the bottommost layer (which has no predecessor). Sometimes, however, the hierarchy is organized in a tree-like structure, which means each layer can have more than one predecessor layer but only one successor layer. Occasionally, a hierarchy can have a many-to-many structure, in which each layer can have multiple predecessors and successors. At no time, shall there be any loop in a hierarchy.

A hierarchy often forms naturally in task decomposition. Often, a task analysis can be decomposed in a hierarchical fashion, starting with the larger tasks and goals of the organization, and breaking each of them down into smaller subtasks that can again be further subdivided. This continuous division of tasks into smaller ones would produce a hierarchical structure of tasks-subtasks.

1.1. Alternate Abstractions

Sometimes it is useful to have multiple alternate abstractions for the same problem so that one can keep different perspectives in mind. For example, we can have a class

diagram, a state chart, and a sequence diagram for the same software at the same level of abstraction. These alternate abstractions do not form a hierarchy but rather complement each other in helping to understand the problem and its solution. Though beneficial, it is as times difficult to keep alternate abstractions in sync.

1. Empirical Methods and Experimental Techniques

[8*, c1]

The engineering process involves proposing solutions or models of solutions and then conducting experiments or tests to study those proposed solutions or models. Thus, engineers must understand how to create an experiment and analyze the results to evaluate proposed solutions. Empirical methods and experimental techniques help the engineer describe and understand variability in their observations, identify the sources of variability, and make decisions.

Three different types of empirical studies commonly used in engineering efforts are designed experiments, observational studies, and retrospective studies. Brief descriptions of the commonly used methods are given.

1.1. Designed Experiment

A designed or controlled experiment is an investigation of a testable hypothesis where one or more independent variables are manipulated to measure their effect on one or more dependent variables. A precondition for conducting an experiment is the existence of a clear hypothesis. It is important for an engineer to understand how to formulate clear hypotheses.

Designed experiments allow engineers to determine in precise terms how the variables are related and, specifically, whether a cause-effect relationship exists between them. Each

combination of values of the independent variables is a *treatment*. The simplest experiments have just two treatments representing two levels of a single independent variable (e.g., using a tool vs. not using a tool). More complex experimental designs arise when more than two levels, more than one independent variable, or any dependent variables are used.

1.1. Observational Study

An observational or case study is an empirical inquiry that makes observations of processes or phenomena within a real-world context. While an experiment deliberately ignores context, an observational or case study includes context as part of the observation. A case study is most useful when the focus of the study is on *how* and *why* questions, when the behavior of those involved in the study cannot be manipulated, and when contextual conditions are relevant and the boundaries between the phenomena and context are not clear.

1.1. Retrospective Study

A retrospective study involves the analysis of historical data. Retrospective studies are also known as historical studies. This type of study uses data (regarding some phenomenon) that has been archived over time. This archived data is then analyzed to find relationships between variables, predict future events, or identify trends. The quality of the analysis results will depend on the quality of the archived data. Historical data may be incomplete, inconsistently measured, or incorrect.

2. Statistical Analysis

[8*, c9s1, c2s1] [9*, c11s3]

Engineers must understand how different product and process characteristics vary.

Engineers often come across situations where the relationship between different variables needs to be studied. An important point to note is that most of the studies are carried out based on samples and so the observed results need to be understood with respect to the full population. Engineers must, therefore, develop an adequate understanding of statistical techniques for collecting reliable data in terms of sampling and analysis to arrive at results that can be generalized. These techniques are discussed.

2.1. Unit of Analysis (Sampling Units), Population, and Sample

Unit of analysis. While carrying out any empirical study, observations need to be made on chosen units called the units of analysis or sampling units. The unit of analysis must be identified and must be appropriate for the analysis. For example, to find the perceived usability of a software product, the user or the software function may be the unit of analysis.

Population. The set of all respondents or items (possible sampling units) to be studied forms the population. As an example, in studying the perceived usability of a software product, the set of all possible users forms the population.

While defining the population, care must be taken to understand the study and target population. The population being studied and the population for which the results are being generalized may be different. For example, when the study population consists of only past observations and generalizations are required for the future, the study population and the target population may not be the same.

Sample. A sample is a subset of the population. The most crucial issue towards the selection of a sample is its representativeness, including size. The

samples must be drawn in a manner that ensures draws are independent, and the rules of drawing samples must be predefined so that the probability of selecting a particular sampling unit is known beforehand. This method of selecting samples is called *probability sampling*.

Random variable. In statistical terminology, the process of making observations or measurements on the sampling units being studied is referred to as conducting the experiment. For example, if the experiment is to toss a coin 10 times and count the number of times the coin lands on heads, each 10 tosses of the coin is a sampling unit and the number of heads for a given sample is the observation or outcome for the experiment. The outcome of an experiment is obtained in terms of real numbers and defines the random variable being studied. Thus, the attribute of the items being measured at the outcome of the experiment represents the random variable being studied; the observation obtained from a particular sampling unit is a particular realization of the random variable. In the example of the coin toss, the random variable is the number of heads observed for each experiment. In statistical studies, attempts are made to understand population characteristics on the basis of samples.

The set of possible values of a random variable may be finite or infinite but countable (e.g., the set of all integers or the set of all odd numbers). In such a case, the random variable is called a *discrete random variable*. In other cases, the random variable under consideration may take values on a continuous scale and is called a *continuous random variable*.

Event. A subset of possible values of a random variable is called an event. Suppose X denotes some random variable; then, for example, we may define different events such as $X \geq x$ or $X < x$ and so on.

Distribution of a random variable. The range and pattern of variation of a random variable is given by its distribution. When the distribution of a random variable is known, it is possible to compute the chance of any event. Some distributions are found to occur commonly and are used to model many random variables occurring in practice in the context of engineering. A few of the more commonly occurring distributions are given below.

- Binomial distribution: used to model random variables that count the number of successes in n trials carried out independently of each other, where each trial results in success or failure. We make an assumption that the chance of obtaining a success remains constant [8*, c3s5].
- Poisson distribution: used to model the count of occurrence of some event over time or space [8*, c3s8].
- Normal distribution: used to model continuous random variables or discrete random variables by taking a very large number of values [8*, c4s5].

Concept of parameters. A statistical distribution is characterized by some parameters. For example, the proportion of success in any given trial is the only parameter characterizing a binomial distribution. Similarly, the Poisson distribution is characterized by a rate of occurrence. A normal distribution is characterized by two parameters: namely, its mean and standard deviation.

Once the values of the parameters are known, the distribution of the random variable is completely known and the chance (probability) of any event can be computed. The probabilities for a discrete random variable can be computed through the Probability Mass Function (PMF). The PMF is defined at discrete points and gives the point mass—i.e., the probability that the

random variable will take that particular value. Likewise, for a continuous random variable, we have the Probability Density Function (PDF). The PDF is very much like density and needs to be integrated over a range to obtain the probability that the continuous random variable lies between certain values. Thus, if the PMF or PDF is known, the chances of the random variable taking certain set of values may be computed theoretically.

Concept of estimation [8*, c7s1, c7s3]. The true values of the parameters of a distribution are usually unknown and need to be estimated from the sample observations. The estimates are functions of the sample values and are called statistics. For example, the sample mean is a statistic and may be used to estimate the population mean. Similarly, the rate of occurrence of defects estimated from the sample (rate of defects per line of code) is a statistic and serves as the estimate of the population rate of rate of defects per line of code. The statistic used to estimate some population parameter is often referred to as the *estimator* of the parameter.

A very important point is that the results of the estimators themselves are random. If we take a different sample, we are likely to get a different estimate of the population parameter. In the theory of estimation, we need to understand different properties of estimators—particularly, how much the estimates can vary across samples and how to choose between different ways to obtain the estimates. For example, if we wish to estimate the mean of a population, we might use as our estimator a sample mean, a sample median, a sample mode, or the midrange of the sample. Each of these estimators has different statistical properties that may impact the standard error of the estimate.

Types of estimates [8*, c7s3, c8s1]. There are two types of estimates: namely, point estimates and interval estimates. When we

use the value of a statistic to estimate a population parameter, we get a point estimate. As the name indicates, a point estimate gives a point value of the parameter being estimated.

Although point estimates are often used, they leave room for many questions. For instance, we are not told anything about the possible size of error or statistical properties of the point estimate. Thus, we might need to supplement a point estimate with the sample size as well as the variance of the estimate. Alternately, we might use an interval estimate. An interval estimate is a random interval with the lower and upper limits of the interval being functions of the sample observations as well as the sample size. The limits are computed based on assumptions regarding the sampling distribution of the point estimate on which the limits are based.

Properties of estimators. Various statistical properties of estimators are used to decide about the appropriateness of an estimator in a given situation. The most important properties are that an estimator is unbiased, efficient, and consistent with respect to the population.

Tests of hypotheses [8*, c9s1]. A hypothesis is a statement about the possible values of a parameter. For example, suppose it is claimed that a new method of software development reduces the occurrence of defects. The hypothesis is that the rate of occurrence of defects has been reduced. In tests of hypotheses, we decide—based on sample observations—whether a proposed hypothesis should be accepted or rejected.

For testing hypotheses, the null and alternative hypotheses are formed. The null hypothesis is the hypothesis of no change and is denoted as H_0 . The alternative hypothesis is written as H_1 . It is important to note that the alternative hypothesis may be one-sided or two-sided. For example, if we have the null hypothesis that the population mean is not

less than some given value, the alternative hypothesis would be that it is less than that value and we would have a one-sided test. However, if we have the null hypothesis that the population mean is equal to some given value, the alternative hypothesis would be that it is not equal and we would have a two-sided test (because the true value could be either less than or greater than the given value).

To test some hypothesis, we first compute some statistic. Along with the computation of the statistic, a region is defined such that if the computed value of the statistic falls in that region, the null hypothesis is rejected. This region is called the critical region (also known as the confidence interval). In tests of hypotheses, we need to accept or reject the null hypothesis based on the evidence obtained. We note that, in general, the alternative hypothesis is the hypothesis of interest. If the computed value of the statistic does not fall inside the critical region, then we cannot reject the null hypothesis. This indicates that there is not enough evidence to believe that the alternative hypothesis is true.

As the decision is being taken based on sample observations, errors are possible; the types of such errors are summarized in the following table.

Nature	Statistical decision	
	Accept H_0	Reject H_0
H_0 is true	OK	Type I error (probability = α)
H_0 is false	Type II error (probability = β)	OK

In test of hypotheses, we aim at maximizing the power of the test (the value of $1-\beta$) while ensuring that the probability of a type I error

(the value of α) is maintained within a particular value—typically 5 percent.

Also note that construction of a test of hypothesis includes identifying statistic(s) to estimate the parameter(s) and defining a critical region such that if the computed value of the statistic falls in the critical region, the null hypothesis is rejected.

2.2. Correlation and Regression

[8*, c11s2, c11s8]

A major objective of many statistical investigations is to establish relationships that make it possible to predict one or more variables in terms of others. Although it is desirable to predict a quantity exactly in terms of another quantity, it is seldom possible and, in many cases, we must be satisfied with estimating the average or expected values.

The relationship between two variables is studied using the methods of correlation and regression. Both these concepts are explained briefly.

Correlation. The strength of linear relationship between two variables is measured using the correlation coefficient. While computing the correlation coefficient between two variables, we assume that these variables measure two different attributes of the same entity. The correlation coefficient takes a value between -1 to $+1$. The values -1 and $+1$ indicate a situation when the association between the variables is perfect—i.e., given the value of one variable, the other can be estimated with no error. A positive correlation coefficient indicates a positive relationship—that is, if one variable increases, so does the other. On the other hand, when the variables are negatively correlated, an increase of one leads to a decrease of the other.

Always remember that correlation does not imply causation. Thus, if two variables are

correlated, we cannot conclude that one causes the other.

Regression. The correlation analysis only measures the degree of relationship between two variables. The analysis to find the relationship between two variables is called *regression analysis*. The strength of the relationship between two variables is measured using the coefficient of determination. This is a value between 0 and 1. The closer the coefficient is to 1, the stronger the relationship between the variables. A value of 1 indicates a perfect relationship.

3. Modeling, Simulation, and Prototyping

[3*, c6] [10*, c13s3] [11*, c5]

Modeling is part of the abstraction process used to represent some aspects of a system. Simulation uses a model of the system and provides a means of conducting designed experiments with that model to better understand the system, its behavior, and relationships between subsystems, as well as to analyze aspects of the design. Modeling and simulation are techniques that can be used to construct theories or hypotheses about the behavior of the system; engineers then use those theories to make predictions about the system. Prototyping is another abstraction process where a partial representation (that captures aspects of interest) of the product or system is built. A prototype may be an initial version of the system but lacks the full functionality of the final version.

1.1. Modeling

A model is always an abstraction of some real or imagined artifact. Engineers use models in many ways as part of their problem solving activities. Some models are physical, such as a made-to-scale miniature construction of a bridge or building. Other models may be

nonphysical representations, such as a CAD drawing of a cog or a mathematical model for a process. Models help engineers reason and understand aspects of a problem. They can also help engineers understand what they do know and what they don't know about the problem at hand.

There are three types of models: iconic, analogic, and symbolic. An iconic model is a visually equivalent but incomplete 2-dimensional or 3-dimensional representation—for example, maps, globes, or built-to-scale models of structures such as bridges or highways. An iconic model actually resembles the artifact modeled.

In contrast, an analogic model is a functionally equivalent but incomplete representation. That is, the model behaves like the physical artifact even though it may not physically resemble it. Examples of analogic models include a miniature airplane for wind tunnel testing or a computer simulation of a manufacturing process.

Finally, a symbolic model is a higher level of abstraction, where the model is represented using symbols such as equations. The model captures the relevant aspects of the process or system in symbolic form. The symbols can then be used to increase the engineer's understanding of the final system. An example is an equation such as $F = Ma$. Such mathematical models can be used to describe and predict properties or behavior of the final system or product.

1.2. Simulation

All simulation models are a specification of reality. A central issue in simulation is to abstract and specify an appropriate simplification of reality. Developing this abstraction is of vital importance, as misspecification of the abstraction would invalidate the results of the simulation exercise. Simulation can be used for a variety of testing purposes.

Simulation is classified based on the type of system under study. Thus, simulation can be either continuous or discrete. In the context of software engineering, the emphasis will be primarily on discrete simulation. Discrete simulations may model event scheduling or process interaction. The main components in such a model include entities, activities and events, resources, the state of the system, a simulation clock, and a random number generator. Output is generated by the simulation and must be analyzed.

An important problem in the development of a discrete simulation is that of initialization. Before a simulation can be run, the initial values of all the state variables must be provided. As the simulation designer may not know what initial values are appropriate for the state variables, these values might be chosen somewhat arbitrarily. For instance, it might be decided that a queue should be initialized as empty and idle. Such a choice of initial condition can have a significant but unrecognized impact on the outcome of the simulation.

1.3. Prototyping

Constructing a prototype of a system is another abstraction process. In this case, an initial version of the system is constructed, often while the system is being designed. This helps the designers determine the feasibility of their design.

There are many uses for a prototype, including the elicitation of requirements, the design and refinement of a user interface to the system, validation of functional requirements, and so on. The objectives and purposes for building the prototype will determine its construction and the level of abstraction used.

The role of prototyping is somewhat different between physical systems and software. With physical systems, the prototype may actually be the first fully functional version of a system or it may be a model of the system. In

software engineering, prototypes are also an abstract model of part of the software but are usually not constructed with all of the architectural, performance, and other quality characteristics expected in the finished product. In either case, prototype construction must have a clear purpose and be planned, monitored, and controlled—it is a technique to study a specific problem within a limited context [12*, c2s8].

In conclusion, modeling, simulation, and prototyping are powerful techniques for studying the behavior of a system from a given perspective. All can be used to perform designed experiments to study various aspects of the system. However, these are abstractions and, as such, may not model all attributes of interest.

4. Measurement

[2*, pp442-447] [3*, c4s4] [12*, c7s5] [13*, c3s1-2]

Knowing what to measure, how to measure, what can be done with measures, and even why to measure is critical in engineering endeavors. It is important that everyone involved in an engineering project understand the measurement methods, measurement results, and how those results can and should be used.

Measurements can be physical, environmental, economic, operational, or some other sort of measurement that is meaningful for the project. This section explores the theory of measurement and how it is fundamental to engineering. Measurement starts as a conceptualization then moves from abstract concepts to definitions of the measurement method to the actual application of that method to obtain a measurement result. Each of these steps must be understood, communicated, and properly employed to yield usable data. In traditional engineering, direct measures are often used.

In software engineering, a combination of both direct and derived measures (defined in 4.3, below) is necessary [13*, p273].

The theory of measurement states that measurement is an attempt to describe an underlying real empirical system. Measurement methods specify activities that assign a value or symbol to an attribute of an entity.

Attributes must then be defined in terms of the operations used to identify and measure them—that is, the *measurement methods*. In this approach, a measurement method is defined to be a precisely specified operation that yields a symbol (called the *measurement result*) when measuring an attribute. It follows that, to be useful, the measurement method must be well defined. Arbitrariness or vagueness in the method will lead to ambiguity in the measurement results.

In some cases—particularly in the physical world—the attributes we wish to measure are easy to grasp; however, in an artificial world like software engineering, defining attributes may not be that simple. For example, the attributes of height, weight, distance, etc. are easily and uniformly understood (though they may not be very easy to measure in all circumstances), whereas attributes such as software size and complexity require clear definitions.

Operational definitions. The definition of attributes, to start with, is often rather abstract. Such definitions do not facilitate measurements. For example, we may define a circle as *a line forming a closed loop such that the distance between any point on this line and a fixed interior point called the center is constant*. We may further say that the fixed distance from the center to any point on the closed loop gives the radius of the circle. It may be noted that though the concept has been defined, no means of

measuring the radius has been proposed. The operational definition specifies the exact steps or method used to carry out a specific measurement. This can also be called the *measurement method*; sometimes a *measurement procedure* may be required to be even more precise.

The importance of operational definitions can hardly be overstated. Take the case of the apparently simple measurement of height of individuals. Unless we specify various factors like the time when the height will be measured (it is known that the height of individuals varies across the day), how the variability due to hair would be taken care of, whether the measurement will be with or without shoes, what kind of accuracy is expected (to the nearest inch, 1/2 inch, centimeter, etc.)—even this simple measurement will lead to substantial variation. Engineers must appreciate the need to define measures from an operational perspective.

4.1. Levels (Scales) of Measurement

[2*, pp442-447] [12*, c7s5] [13*, c3s2]

Once the operational definitions have been determined, actual measurements can be undertaken. Measurement may be carried out in one of four different scale types: nominal, ordinal, interval, and ratio. Brief descriptions of each are given below.

Nominal scale: This is the lowest level of measurement and represents the most unrestricted assignment of symbols, which serve only as labels. Nominal scales involve only classification where measured entities are put into one of the mutually exclusive and collectively exhaustive categories (classes). Some examples of nominal scales are:

- job titles in an organization;
- automobile styles (like sedan, coupe, hatchback, minivan, etc.);

- software development life cycle (SDLC) models (like waterfall, iterative, agile, etc.).

In nominal scales, no relationship between symbols may be inferred. The only valid manipulation of measures in a nominal scale are:

- determining if two entities have the same or different symbol (e.g., “Is your job title the same or different than my job title?”);
- counting the number of entities having the same symbol (e.g., “How many employees have job title Software Engineer Level 2 in this organization?”).

Statistical analyses may be carried out to understand how entities belonging to different classes perform with respect to some other response variable.

Ordinal scale: Ordinal scales extend nominal scales by requiring a strict ordering relationship between the symbols. Ordinal scales are necessarily transitive, if $A > B$ and $B > C$, then $A > C$. Examples of ordinal scales are:

- finish order in a race (e.g., 1st, 2nd, 3rd, etc.)
- probabilities expressed using terms: remote, unlikely, even, probable, and almost certain;
- severities expressed using terms: negligible, marginal, serious, critical, and catastrophic;
- level of agreement expressed using terms: strongly agree, somewhat agree, neutral, somewhat disagree, strongly disagree;
- Capability Maturity Model Integration (CMMI) staged maturity levels.

All manipulations of values on nominal scales are valid on ordinal scales, while

ordinal scales also support more-than and less-than comparisons. For example:

- did you finish that race before, tied with, or after, me?
- is Event X the same, more, or less, probable than Event Y?
- is Event X the same, more, or less, severe than Event Y?
- is the CMMI staged maturity level of Organization A the same, higher, or lower, than Organization B?

When an ordinal scale uses numbers as symbols—like the CMMI staged maturity levels 1, 2, 3, 4, and 5—those numbers cannot be manipulated arithmetically. We cannot say that the difference between CMMI staged maturity level 5 and level 4 (i.e., 5-4) compares in any meaningful way to the difference between level 3 and level 2 (i.e., 3-2). Neither can we say that CMMI staged maturity level 4 is twice as good as level 2. Ordinal scales that use numbers as symbols are commonly misused in exactly this way, for example presenting mean and standard deviation (e.g., “The average software organization worldwide has a CMMI staged maturity level of 1.763”). Such misuse can easily lead to erroneous conclusions [13*, p274]. We can compute the median on an ordinal scale as this only involves counting. Using non-numeric symbols, such as initial, repeatable, defined, managed, and optimizing for CMMI staged maturity levels, is preferred because it helps avoid such mistreatment. Properly chosen labels also better communicate the meaning of each label.

Interval scale: Interval scales extend ordinal scales by requiring that the difference between any pair of adjacent values is constant. Examples of interval scales are:

- temperatures expressed in degrees Celsius and Fahrenheit. The difference between -9C and -8C is the same as the difference between 26C and 27C. The difference between -9F and -8F is the same as the difference between 26F and 27F;
- calendar dates. The difference between any two consecutive dates is always one day: 24 hours;
- shoe sizes in North America. The difference between size 3 and size 4 is the same as the difference between a size 10 and size 11, namely one third of an inch or 8.467mm.

All manipulations of values on ordinal scales are valid on interval scales, while interval scales also support addition and subtraction. For example:

- the difference between -9C and 0C is the same as the difference between 0C and 9C. The difference between -50F and -0F is the same as the difference between 25F and 75F;
- the length of time between the 6th of May and the 9th of May is the same as between the 8th of November and the 11th of November.

Interval scales support most of the usual statistical analyses like mean, standard deviation, correlation, and regression. Any manipulation involving multiplication or division of values, on the other hand, is meaningless because zero on an interval scale, if it even exists, does not represent absence of the measured quantity. A zero point on an interval scale is arbitrary with respect to the attribute measured. Zero degrees both C and F do not represent the absence of heat (i.e., absolute zero) and a North American size 0 shoe has non-zero length. Therefore, 30C cannot be interpreted as twice as hot as 15C, nor is a North American size 9 shoe three times longer than a size 3 shoe.

Ratio scale: Ratio scales extend ordinal scales by requiring the zero point represent absence of the measured attribute. Examples of ratio scales are:

- temperature in degrees Kelvin (K);
- shoe sizes in the Mondopoint system (commonly used for athletic shoes, ski boots, skates, and ballet shoes). A size 270/105 shoe fits a foot 270mm long and 105mm wide;
- count of decision constructs, e.g., if(), for(), while(), in a given source code file;
- money.

Ratio scales support all arithmetic and statistical manipulations. Values in one ratio scale can often be trivially transformed into corresponding values in another ratio scale that measures the same attribute by using a multiplication factor. Distances in inches can be trivially transformed into centimeters, weights in kilograms can be trivially transformed into pounds, speed in knots can be trivially transformed into kilometers per hour, and so on.

An additional measurement scale, the *absolute scale*, is a ratio scale with uniqueness of measure, i.e., no transformation is possible. The number of software engineers working on a project is an absolute scale because there are no other meaningful measures for numbers of people.

1.1. Implications of Measurement Theory on Programming Languages

Common programming languages support a set of built-in data types, often including:

- whole number types over varying ranges: int, integer, byte, short, long, etc.;

- floating point numbers over varying ranges with varying precision: real, float, double, etc.;
- single characters: char;
- ordered sequences of characters: String.

Many languages, although not all, also support type safe enumeration (e.g., Java’s “enum”).

Unfortunately, these languages offer no support for Measurement Theory. They do not prevent, nor even warn about, inappropriate manipulations. The whole and floating-point number data types in programming languages operate as ratio scales and support the full range of manipulations: comparison, addition, subtraction, multiplication, division, and so on. But consider CMMI staged maturity level expressed as a number. In Measurement Theory terms, as shown above, it is an Ordinal scale so addition, subtraction, multiplication, and division are inappropriate. If any programmer represents CMMI staged maturity level using a whole number data type, nothing prevents them from inappropriately adding, subtracting, multiplying, or dividing.

The same can be said for characters, strings, and enumerations. Programming languages implement them as Ordinal scales however they might only be intended for representing Nominal scaled values. More-than and less-than comparisons are allowed even when inappropriate. The string, “minivan” is alphabetically before the string “sedan”, but it is inappropriate to draw any conclusion other than mere alphabetical ordering of arbitrary text strings out of that fact.

Common programming languages allow programmers to easily write code that is inappropriate according to Measurement Theory. As long as programming languages allow it, programmers can and will—intentionally or unintentionally—misuse them. A more sensible solution would be data type semantics that explicitly enforce Measurement Theory. For example, a language could explicitly support Nominal scales, such as,

```
nominal enum automobile_style
= sedan, coupe, hatchback,
minivan, suv, sports_car;
```

That language could then prevent, or at least warn against, more-than or less-than comparisons:

```
if( thisCarStyle >= sedan )
then ... // not allowed
```

If more-than or less-than comparisons are needed, the language would support declaration of an Ordinal type, such as,

```
ordinal enum CMMI_staged_level
= initial, repeatable,
defined, managed, and
optimizing;
```

The following statement would not trigger any error or warning:

```
if( anOrgsCMMILevel      >
repeatable ) then ...
```

Similarly, an interval scale could be supported as, for example:

```
interval
AirTemperatureCentigrade from
-120.0 to +180.0;
```

```
AirTemperatureCentigrade
yesterdaysHighTemp;
AirTemperatureCentigrade
todaysHighTemp;
```

```
if(      todaysHighTemp      >
yesterdaysHighTemp ) { ... } // allowed
```

```
if(      todaysHighTemp      >
yesterdaysHighTemp * 2.0 ) { ... } // not allowed
```

A ratio scale could be supported as, for example:

```
ratio TemperatureKelvin from
0.00 to 1000.00;
```

```
TemperatureKelvin
previousReading;
TemperatureKelvin thisReading;
```

```
if(      thisReading      >
previousReading * 2. ) { ... } // allowed
```

Common programming languages have no problem with the following code:

```
double priceOfBook;
double highTemperature;
```

```
highTemperature = priceOfBook;
// makes no sense but is allowed
```

On the other hand, a Measurement Theory-aware programming language would be expected to generate a compiler error or warning:

```
ratio Money from -10000.00 to
+10000.00;
ratio TemperatureKelvin from
0.00 to 1000.00;
Money priceOfBook;
TemperatureKelvin
highTemperature;
```

```
highTemperature = priceOfBook;
// not allowed
```

Future programming languages should enforce Measurement Theory and not allow developers to manipulate measurements in inappropriate ways. But until languages do support Measurement Theory, software engineers need to at least understand it and be on the lookout for inappropriate manipulations in, for example, code reviews.

1.1. Direct and Derived Measures

[13*, c7s5]

Measures may be either direct or derived (sometimes called indirect measures). An example of a direct measure would be a count of how many times an event occurred, such as the number of defects found in a software product. A derived measure is one that combines direct measures in some way that is consistent with the measurement methods. An example of a derived measure would be

calculating the average effort in hours to repair defects found in a software product. In both cases, the measurement method determines how to make the measurement. The scale types of those measures constrain how they can be manipulated. When different scale types are involved:

- the scale type of the result of the manipulation can be no higher than the scale type of the most primitive measurement scale, e.g., a manipulation involving an interval scale and a ratio scale can only be done as if both are interval scales and yields no better than an interval scale result;
- investment is required to bring the most primitive scale type up to being compatible with a higher scale type, e.g., effort is required to bring the interval scale up to also being a ratio scale so that the result can also be ratio scaled.

1.1. Reliability and Validity

[13*, c3s4-5]

A basic question to be asked for any measurement method is whether the proposed measurement method is truly measuring the concept with good quality. Reliability and validity are the two most important criteria to address this question.

The reliability of a measurement method is the extent to which the application of the measurement method yields consistent measurement results. Essentially, *reliability* refers to the consistency of the values obtained when the same item is measured several times. When the results agree with each other, the measurement method is said to be reliable. Reliability usually depends on the operational definition. It can be quantified by using the index of variation, which is computed as the ratio between the standard deviation and the mean. The smaller the index, the more reliable the measurement results.

Validity refers to whether the measurement method really measures what we intend to measure. Validity of a measurement method may be looked at from three different perspectives: construct validity, criteria validity, and content validity.

1.1. Assessing Reliability

[13*, c3s5]

There are several methods for assessing reliability; these include the test-retest method, the alternative form method, the split-halves method, and the internal consistency method. The easiest of these is the test-retest method. In the test-retest method, we simply apply the measurement method to the same subjects twice. The correlation coefficient between the first and second set of measurement results gives the reliability of the measurement method.

1.1. Goal-Question-Metric Paradigm: Why Measure?

The final concern about measurement is to understand why we are measuring in the first place. The Software Process KA chapter <<<TBD—should: reference the section if so>>> discuss the Goal-Question-Metric paradigm in more detail, but it can be summarized with the simple observation that a measurement should exist to support making one or more decisions. Some measurements support decisions in code. Other measurements support decisions made by people outside of code, e.g., process improvement measures. The critical point is that some decision be made on that measurement. Many software metrics programs in real-world software organizations fall victim to a “measurement for the merely curious” syndrome, where metrics are gathered simply because they are easy to measure and interesting to look at when plotted in graphs. Those measurements

are never used to support any decision and are a waste of time and energy. They should be avoided.

5. Standards

[3*, c9s3.2]

Moore states that a

standard can be; (a) an object or measure of comparison that defines or represents the magnitude of a unit; (b) a characterization that establishes allowable tolerances for categories of items; and (c) a degree or level of required excellence or attainment. Standards are definitional in nature, established either to further understanding and interaction or to acknowledge observed (or desired) norms of exhibited characteristics or behavior. [14, p8]

Standards provide requirements, specifications, guidelines, or characteristics that must be obeyed by engineers so that the products, processes, and materials have acceptable levels of quality. The qualities that various standards provide may be those of safety, reliability, or other product characteristics. Standards are considered critical to engineers and engineers are expected to be familiar with and to use the appropriate standards in their discipline.

Compliance or conformance to a standard allows an organization say to the public that they (or their products) meet the requirements stated in that standard. Thus, standards divide organizations or their products into those that conform to the standard and those that do not. For a standard to be useful, conformance must add value—real or perceived—to the product, process, or effort.

Apart from the organizational goals, standards are used for several other purposes

such as protecting the buyer, protecting the business, and better defining the methods and procedures to be followed by the practice. Standards also provide users with a common terminology and expectations.

There are many internationally recognized standards-making organizations including the International Telecommunications Union (ITU), the International Electrotechnical Commission (IEC), IEEE, and the International Organization for Standardization (ISO). In addition, there are regional and governmentally recognized organizations that generate standards for that region or country. For example, in the United States, there are over 300 organizations that develop standards. These include organizations such as the American National Standards Institute (ANSI), the American Society for Testing and Materials (ASTM), the Society of Automotive Engineers (SAE), and Underwriters Laboratories, Inc. (UL), as well as the US government. For more detail on standards used in software engineering, see Appendix B on standards.

There is a set of commonly used principles behind standards. Standards makers attempt to have consensus around their decisions. There is usually an openness within the community of interest so that once a standard has been set, there is a good chance that it will be widely accepted. Most standards organizations have well-defined processes for their efforts and adhere to those processes carefully. Engineers must be aware of the existing standards but must also update their understanding of the standards as those standards change over time.

In many engineering endeavors, knowing and understanding the applicable standards is critical and the law may even require use of specific standards. In these cases, the standards often represent minimal requirements that must be met by the endeavor and thus are an element in the

constraints imposed on any design effort. The engineer must review all current standards related to a given endeavor and determine which must be met. Their designs must then incorporate any and all constraints imposed by the applicable standard.

6. Root Cause Analysis

[3*, c9s3-5] [13*, c5, c3s7, c9s8]

Root cause analysis (RCA) is a class of problem-solving methods aimed at identifying underlying causes of undesirable outcomes. RCA methods identify why and how an undesirable outcome happened, allowing organizations to take effective actions to prevent recurrence. Instead of merely addressing immediately obvious symptoms, problems can be solved by eliminating root causes. RCA can play several important roles on software projects, including:

- identify the real problem to be solved by an engineering effort;
- expose the underlying drivers of risk when performing project risk assessments;
- reveal opportunities and actions for software process improvement;
- discover sources of recurring defects (i.e., defect causal analysis).

6.1. Root Cause Analysis Techniques

Several RCA techniques exist, including:

- Change Analysis compares situations where an undesirable outcome happened with similar situations where it did not. The root cause is likely in the area of difference;
- 5-Whys (see, for example, [2*, c4]) starts with an undesirable outcome and uses repeated “Why?” question-answer cycles to isolate root cause;

- Cause-and-Effect diagrams, sometimes called Ishikawa diagrams [15] or Fishbone charts, break down, in successive levels of detail, causes that potentially contribute to an undesirable outcome. Causes are often grouped into major categories such as people, process, tools, materials, measurements, and environment. The diagram takes the form of a tree of potential causes that can all result in that undesirable outcome;

- Fault Tree Analysis (FTA) is a more formal approach to cause-and-effect diagramming which is more specific about and-or relationships between causes and effects. In some cases, any one of multiple causes can drive the effect (an “or” relationship), in other cases a combination of multiple causes are required to drive the effect (an “and” relationship). Cause-and-effect diagrams do not distinguish between and-or relationships, Fault tree analysis does;

- Failure Modes and Effects Analysis (FMEA) forward-chains, starting with elements that can fail and cascades into the undesirable effects that could result from those failures. This contrasts with the backwards-chaining approaches above that start from an undesirable outcome and work backwards toward causes;

- Cause Map [16] is a structured map of cause-effect relationships that includes an undesirable outcome along with 1) chaining backwards to driving causes and 2) chaining forward into effects on organizational goals. Cause maps demand evidence of the occurrence of causes and the causality of effects and are thus more rigorous than Cause-and-Effect diagrams, FTA, and FMEA;

- Current Reality Tree [17] is a cause-effect tree bound by the rules of logic (i.e., Categories of Legitimate Reservation);

- Human Performance Evaluation posits that human performance depends on 1) input

detection, 2) input understanding, 3) action selection, and 4) action execution. An undesirable outcome that results from human performance can be identified from a comprehensive list of potential drivers that includes, for example, cognitive overload, cognitive underload (i.e., boredom), memory lapse, tunnel vision or lack of a bigger picture, complacency, fatigue, etc.

Additional techniques can be found in DOE-NE-STD-1004-92 Root Cause Analysis Guidance Document.

1.1. Root Cause-Based Improvement

RCA is often an element in a larger process improvement effort. Why just identify a root cause if nothing will be done about it? Why go through the effort of identifying root cause on low importance problems? An example of a systematic process for a larger improvement effort incorporating RCA is given below.

1. Select the problem to solve: techniques such as Pareto Analysis (i.e., the “80/20 Rule”), Frequency-Severity Prioritization (problems that happen most frequently and consume the most resources to rectify are the best candidates), Statistical Process Control, etc. are used to identify a high priority undesirable outcome to address. This step needs to clearly define the problem and its significance.

2. Gather evidence about that problem and its cause(s): consider information surrounding the selected undesirable outcome including, for example, statements or testimony, relevant processes or standards, specifications, reports, historical trends, experiments, or tests.

3. Identify the root cause using one or more of the RCA techniques presented in X.1.

4. Select corrective action(s) that 1) prevent recurrence, 2) are within the organization’s ability to control, 3) meet organizational goals and objectives, 4) do not cause other problems. More than one candidate corrective action should be considered that all either prevent the cause from happening, reduce the probability of the cause happening, or disconnect the cause from the effect. Selected corrective actions should generate the greatest amount of control for the least cost.

5. Implement the selected corrective action(s).

6. Observe the selected corrective action(s) to ensure efficiency and effectiveness.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	Tockey 2004 [2*]	Voland 2003 [3*]	McConne ll 2004 [6*]	Montgomery and Rung er 2018 [8*]	Null and Lobur 2006 [9*]	Che ney and Kin caid 2007 [10*]	Sommerville 2018 [11*]	Fairley 2009 [12*]	Kan 2002 [13*]
1. The Engineering Process	c4								
2. Engineering Design		c1s2-4							
2.1 Design in Engineering Education									
2.2 Design as a Problem Solving Activity		c1s4, c2s1, c3s3	c5s1						
3. Abstraction and Encapsulation			c5s2-4						
3.1 Levels of Abstraction									
3.2 Encapsulation									
3.3 Hierarchy									
3.4 Alternate Abstractions									
4. Empirical Methods and Experimental Techniques				c1					
4.1 Designed Experiment									
4.2 Observational Study									
4.3 Retrospective Study									
5. Statistical Analysis				c9s1, c2s1	c11s3				
5.1 Unit of Analysis (Sampling Units),				c3s5, c3s8, c4s5, c7s1, c7s3,					

Population, and Sample				c8s1, c9s1					
5.2 Correlation and Regression				c11s2, c11s8					
6. Modeling, Prototyping, and Simulation		c6			c13s3	c5			
6.1 Modeling									
6.2 Simulation									
6.3 Prototyping							c2s8		
7. Measurement	pp442-447	c4s4					c7s5	c3s1-2	
7.1 Levels (Scales) of Measurement	p442-447						c7s5	c3s2	
7.2 Implications of Measurement Theory on Programming Languages									
7.3 Direct and Derived Measures								c7s5	
7.4 Reliability and Validity								c3s4-5	
7.5 Assessing Reliability								c3s5	
7.6 Goal-Question-Metric Paradigm: Why Measure?								c3s5	
8. Standards		c9s3.2							
9. Root Cause Analysis		c9s3-5						c5, c3s7, c9s8	
9.1 Root Cause Analysis Techniques	c4							c5	
9.1 Root Cause-Based Improvement	c4							c5	

FURTHER READINGS

A. Abran, *Software Metrics and Software Metrology*. [18]

This book provides very good information on the proper use of the terms measure,

measurement method and measurement outcome. It provides strong support material for the entire section on Measurement.

W.G. Vincenti, *What Engineers Know and How They Know It*. [19]

This book provides an interesting introduction to engineering foundations through a series of case studies that show many of the foundational concepts as used in real world engineering applications.

REFERENCES

- [1] IEEE/ISO/IEC, "IEEE/ISO/IEC 24765: Systems and Software Engineering - Vocabulary," 1st ed, 2010.
- [2*] S. Tockey, *Return on Software: Maximizing the Return on Your Software Investment*, 1st ed. Boston: Addison-Wesley, 2004.
- [3*] G. Voland, *Engineering by Design*, 2nd ed. Upper Saddle River, NJ: Prentice Hall, 2003.
- [4] "2021 Accreditation Criteria and Procedures," Canadian Engineering Accreditation Board, Engineers Canada, 2021.
- [5] E. A. Commission, "Criteria for Accrediting Engineering Programs, 2022-2023," ABET, 2021.
- [6*] S. McConnell, *Code Complete*, 2nd ed. Redmond, WA: Microsoft Press, 2004.
- [7] Edsgar W. Dijkstra, The Humble Programmer, *Communications of the ACM*, vol 15, issue 10, October, 1972.
- [8*] D. C. Montgomery and G. C. Runger, *Applied Statistics and Probability for Engineers*, 7th ed. Hoboken, NJ: Wiley, 2018.
- [9*] L. Null and J. Lobur, *The Essentials of Computer Organization and Architecture*, 5th ed. Sudbury, MA: Jones and Bartlett Publishers, 2018.
- [10*] E. W. Cheney and D. R. Kincaid, *Numerical Mathematics and Computing*, 6th ed. Belmont, CA: Brooks/Cole, 2007.
- [11*] I. Sommerville, *Software Engineering*, 10th ed. New York: Addison-Wesley, 2018.
- [12*] R. E. Fairley, *Managing and Leading Software Projects*. Hoboken, NJ: Wiley-IEEE Computer Society Press, 2009.
- [13*] S. H. Kan, *Metrics and Models in Software Quality Engineering*, 2nd ed. Boston: Addison-Wesley, 2002.
- [14] J. W. Moore, *The Road Map to Software Engineering: A Standards-Based Guide*, 1st ed. Hoboken, NJ: Wiley-IEEE Computer Society Press, 2006.
- [15] K. Ishikawa, *Introduction to Quality Control*, Productivity Press, 1990.
- [16] D. Gano, *Apollo Root Cause Analysis*, 3rd Ed., Apollonian Publications, 2007.
- [17] E. Goldratt, *It's Not Luck*, North River Press, 1994.
- [18] A. Abran, *Software Metrics and Software Metrology*: Wiley-IEEE Computer Society Press, 2010.
- [19] W. G. Vincenti, *What Engineers Know and How they Know It*: John Hopkins University Press, 1993.

APPENDIX A

KNOWLEDGE AREA DESCRIPTION SPECIFICATIONS

INTRODUCTION

This appendix presents the specifications provided to the Knowledge Area (KA) editors regarding the KA Descriptions of the *Guide to the Software Engineering Body of Knowledge, Version 4 (SWEBOk Guide, V4)*. This enables readers, reviewers and users to clearly understand what specifications were used in developing this version of the *SWEBOk Guide*.

This appendix begins by situating the *SWEBOk Guide* as a foundational document for the IEEE Computer Society's suite of software engineering products and more widely within the software engineering community. The appendix then describes the role of the baseline and the Change Control Board. Criteria and requirements are defined for the breakdowns of topics, for the rationale underlying these breakdowns and the succinct description of topics, and for reference materials. Important input documents are also identified, and their role within the project is explained. Finally, non-content issues such as submission format and style guidelines are discussed.

THE SWEBOK GUIDE IS A FOUNDATIONAL DOCUMENT FOR THE IEEE COMPUTER SOCIETY SUITE OF SOFTWARE ENGINEERING PRODUCTS

The *SWEBOk Guide* is an IEEE Computer Society flagship and structural document for the IEEE Computer Society's suite of software engineering products. The *SWEBOk Guide* is also more widely recognized as a foundational document throughout the software engineering community, notably through the official recognition of the 2004 and 2014 versions as ISO/IEC Technical Report 19759:2005 and 19759:2015, respectively. The list of KAs and the breakdown of topics within each are described and detailed in this *SWEBOk Guide*'s introduction. Consequently, the *SWEBOk Guide* is foundational to other initiatives within the IEEE Computer Society, as follows:

- The list of KAs and the breakdown of topics within each are also adopted by the software engineering certification and associated professional development products offered by the IEEE Computer Society. (See www.computer.org/certification.)
- The list of KAs and the breakdown of topics are also foundational to the software engineering curriculum guidelines developed or endorsed by the IEEE Computer Society. (See www.computer.org/portal/web/education/Curricula.)
- The Consolidated Reference List (see Appendix C) — meaning the list of Recommended References (to the level of section number) that accompanies the breakdown of topics within each KA —

is also adopted by the software engineering certification and associated professional development products offered by the IEEE Computer Society.

BASELINE AND CHANGE CONTROL

Due to the structural nature of the *SWEBOK Guide* and its adoption by other products, a baseline was developed at the outset of the project by a SWEBOK Steering Group. The baseline comprises the list of KAs, including new ones, and the breakdown of topics for each KA from the previous version.

Furthermore, a *SWEBOK* KA editors team has been put in place for the development of this version to handle all major change requests to this baseline coming from the KA editors, arising during the review process or otherwise. Change requests must be approved both by the *SWEBOK Guide* editors and by the team before being implemented. The team is composed of members of the initiatives listed above and acts under the authority of the Engineering Discipline Committee of the IEEE Computer Society Professional and Educational Activities Board (PEAB).

CRITERIA AND REQUIREMENTS FOR THE BREAKDOWN OF TOPICS WITHIN A KNOWLEDGE AREA

- KA editors are instructed to refine the baseline breakdown of topics to reflect the recent development in the target area for KAs that continue to exist from the previous version.
- The breakdown of topics is expected to be “reasonable,” not “perfect.”
- The breakdown of topics within a KA must decompose the subset of the *SWEBOK* that is “generally recognized.” (See below for a more detailed discussion of this point.)
- The breakdown of topics within a KA must not presume specific application domains, business needs, sizes of organizations, organizational structures, management philosophies, software life cycle models, software technologies or software development methods.
- The breakdown of topics must, as much as possible, be compatible with the various schools of thought within software engineering.
- The breakdown of topics within a KA must be compatible with the breakdown of software engineering generally found in industry and in the software engineering literature and standards.
- The breakdown of topics is expected to be as inclusive as possible.
- The *SWEBOK Guide* adopts the position that even though the following “themes” are common across all KAs, they are also an integral part of all KAs and, therefore, must be incorporated into the proposed breakdown of topics of each KA. These common themes are measurement, quality (in general) and security.
- The breakdown of topics should be at most two or three levels deep. Even though no upper or lower limit is imposed on the number of topics within each KA, a reasonable and manageable number of topics is expected to be included in each KA. Emphasis should also be put on the selection of the topics themselves rather than on their organization in an appropriate hierarchy.
- Topic names must be significant enough to be meaningful even when cited outside the *SWEBOK Guide*.
- The Description of a KA will include a chart (in tree form) describing the knowledge breakdown. This chart will typically be the first figure in the respective KA.

CRITERIA AND REQUIREMENTS FOR DESCRIBING TOPICS

Topics need only be sufficiently described so readers can select the appropriate reference material according to their needs. Topic descriptions must not be prescriptive.

CRITERIA AND REQUIREMENTS FOR REFERENCE MATERIAL

- KA editors are instructed to use the references (to the level of section number) allocated to their KA by the Consolidated Reference List as their Recommended References.
- There are three categories of reference material:
 1. Recommended References. The set of Recommended References (to the level of section number) is collectively known as the Consolidated Reference List.
 2. Further Reading.
 3. Additional references cited in the KA Description (e.g., the source of a quotation or reference material in support of a rationale behind a particular argument).
- The *SWEBOK Guide* is intended by definition to be selective in its choice of topics and associated reference material. The list of reference material should be clearly viewed as an “informed and reasonable selection” rather than as a definitive list.
- Reference material can be book chapters, refereed journal papers, refereed conference papers, refereed technical or industrial reports, or any other type of recognized artifact. References to another KA, subarea or topic are also permitted.
- Reference material must be generally available and must not be confidential in nature.
- Reference material must be in English.

- Criteria and requirements for recommended reference material or Consolidated Reference List:
 - » Collectively, the list of Recommended References should be:
 - i. Complete — covering the entire scope of the *SWEBOK Guide*
 - ii. Sufficient — providing enough information to describe “generally accepted” knowledge
 - iii. Consistent — not providing contradictory knowledge or conflicting practices
 - iv. Credible — recognized as providing expert treatment
 - v. Current — treating the subject in a manner that is commensurate with current, generally accepted knowledge
 - vi. Succinct — as short as possible (both in the number of reference items and in total page count) without failing other objectives
 - » Recommended reference material must be identified for each topic. Each recommended reference item may, of course, cover multiple topics. Rarely, a topic may be self-descriptive and not cite a reference material item (e.g., a topic that is a definition or a topic for which the description itself without any cited reference material is sufficient for the objectives of the *SWEBOK Guide*).
 - » Each reference to the recommended reference material should be as precise as possible, identifying what specific chapter or section is relevant.
 - » A matrix of reference material (to the level of section number) versus topics must be provided.
 - » The latest versions or editions should be used as the Recommended References if there are multiple versions or editions.

- » A reasonable amount of recommended reference material must be identified for each KA. The following guidelines should be used in determining how much is reasonable:
 - i. If the recommended reference materials are written in a coherent manner, follow the proposed breakdown of topics, and use a consistent style (e.g., list a new book based on the proposed KA description), an average page number target across all KAs would be 750. However, this target may not be attainable when selecting existing reference material due to differences in style and to overlap and redundancy among the selected reference materials.
 - ii. In other words, the target for the number of pages for the entire collection of Recommended References in the *SWEBOK Guide* is in the range of 10,000 to 15,000 pages.
 - iii. Another way of viewing this is that the amount of recommended reference material would be reasonable if it consisted of the study material for this KA for a software engineering licensing exam that a graduate would pass after completing four years of work experience.
- Additional reference material can be included by the KA editor in a “Further Reading” list:
 - » These materials must be related to the topics in the breakdown rather than, for example, to more advanced topics.
 - » The list must be annotated (one paragraph per reference) to explain why each reference was included. Further Reading could include alternative viewpoints on a KA or a seminal treatment of a KA.
 - » A general guideline to be followed is 10 or fewer further readings per KA.
 - » There is no matrix of the reference materials listed in Further Reading and the breakdown of topics.
- Criteria and requirements regarding additional references cited in the KA Description:
 - » The *SWEBOK Guide* is not a research document, and its readership will be varied. Therefore, a delicate balance must be maintained between ensuring a high level of readability within the document and maintaining its technical excellence. Additional reference material should, therefore, be brought in by the KA editor only if it is necessary to the discussion. For example, the reference material might identify the source of a quotation or offer support for the rationale behind an important argument.

COMMON STRUCTURE

KA Descriptions should use the following structure:

- Abbreviations
- Introduction
- Breakdown of Topics of the KA (including a figure describing the breakdown)
- Matrix of Topics vs. Reference Material
- List of Further Reading
- References

WHAT DO WE MEAN BY “GENERALLY RECOGNIZED KNOWLEDGE”?

The Software Engineering Body of Knowledge is an all-inclusive term that describes the sum of knowledge within the profession of software engineering. However, the *SWEBOk Guide* seeks to identify and describe that subset of the body of knowledge that is generally recognized or, in other words, the core body of knowledge. To better illustrate what “generally recognized” knowledge is relative to other types of knowledge, Figure A.1 proposes a three-category schema for classifying knowledge.

The Project Management Institute, in its *Guide to the Project Management Body of Knowledge*, defines “generally recognized” knowledge for project management as:

that subset of the project management body of knowledge generally recognized as good practice. “Generally recognized” means the knowledge and practices described are applicable to most projects most of the time, and there is consensus about their value and usefulness. “Good practice” means there is general agreement that the application of these skills, tools, and techniques can enhance the chances of success over a wide range of projects. “Good practice” does not mean that the knowledge described should always be applied uniformly to all projects; the organization and/or project management team is responsible for determining what is appropriate for any given project [1].

“Generally accepted” knowledge could also be viewed as knowledge to be included in the study material of a software engineering licensing exam (in the US) that a graduate would take after completing four years of work experience. These two definitions should be seen as complementary.

KA editors are also expected to be somewhat forward-looking in their interpretation by taking into consideration not only what is “generally recognized” today but also what they expect will be “generally recognized” in a three- to five-year time frame.

Practices	Generally Recognized Established traditional practices recommended by many organizations
Special	Advanced and Research Innovative practices tested and used only by some organizations and concepts still being developed and tested in research organizations

Figure A.1. Categories of Knowledge

LENGTH OF KA DESCRIPTION

KA Descriptions are to be roughly 10 to 20 pages using the formatting template for papers published in conference proceedings of the IEEE Computer Society. This includes text, references, appendixes, tables, etc. This, of course, does not include the reference materials themselves.

IMPORTANT RELATED DOCUMENTS

1. *Graduate Software Engineering 2009 (GSwE2009): Curriculum Guidelines for Graduate Degree Programs in Software Engineering*, 2009 [2].

This document “provides guidelines and recommendations” for defining the curricula of a professional master’s-level program in software engineering. The *SWEBOk Guide* is identified as a “primary reference” in

developing the body of knowledge underlying these guidelines. This document has been officially endorsed by the IEEE Computer Society and sponsored by the Association for Computing Machinery.

2. ISO/IEC/IEEE 12207-
2017 Standard for Systems and Software Engineering — Software Life Cycle Processes, ISO/IEC/IEEE, 2017 [3].

This standard is considered the key standard regarding the definition of life cycle processes and has been adopted by the two main standardization bodies in software engineering: ISO/IEC JTC1/SC7 and the IEEE Computer Society Software and Systems Engineering Standards Committees. It also has been designated a pivotal standard by the Software and Systems Engineering Standards Committee (S2ESC) of the IEEE.

Even though we do not intend the *SWEBOK Guide* to be fully 12207-conformant, this standard remains a key input to the *SWEBOK Guide*, and special care will be taken throughout the *SWEBOK Guide* regarding the compatibility of the *Guide* with the 12207 standard.

3.
“Software Engineering 2014: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering,” IEEE Computer Society and Association for Computing Machinery, 2015; <https://www.acm.org/binaries/content/assets/education/sc2014.pdf> [4].

This document describes curriculum guidelines for an undergraduate degree in software engineering. The *SWEBOK Guide* is identified as “one of the primary sources” in developing the body of knowledge underlying these guidelines.

4. “ISO/IEC/IEEE 24765:2017 Software and Systems Engineering — Vocabulary,” ISO/IEC/IEEE, 2017; <https://www.computer.org/sevocab> [5].

The hierarchy of references for terminology is *Merriam-Webster’s Collegiate Dictionary* (11th ed.) [6], IEEE/ISO/IEC 24765 [5] and newly proposed definitions, if required.

5. “Software Professional Certification Program,” IEEE Computer Society; <https://www.computer.org/education/certifications> [7].

Information on the certification and associated professional development products developed and offered by the IEEE Computer Society for professionals in the field of software engineering can be found on this website. The *SWEBOK Guide* is foundational to these products.

OTHER DETAILED GUIDELINES

When referencing the *Guide to the Software Engineering Body of Knowledge*, use the title *SWEBOK Guide*.

For the purpose of simplicity, avoid footnotes, and try to include their content in the main text.

Use explicit references to standards, as opposed to simply inserting numbers referencing items in the bibliography. We believe this approach allows the reader to be better exposed to the source and scope of a standard.

The text accompanying figures and tables should be self-explanatory or have enough related text. This ensures that the reader knows what the figures and tables mean.

To make sure that some information in the *SWEBOK Guide* does not become rapidly obsolete and in order to reflect its generic

nature, please avoid directly naming tools and products. Instead, try to name their functions.

EDITING

Editors of the *SWEBOk Guide*, as well as professional copy editors, will edit KA Descriptions. Editing includes copy editing (grammar, punctuation and capitalization), style editing (conformance to the Computer Society style guide), and content editing (flow, meaning, clarity, directness and organization). The final editing will be a collaborative process in which the editors of the *SWEBOk Guide* and the KA editors will work together to achieve a concise, well-worded and useful KA Description.

RELEASE OF COPYRIGHT

All intellectual property rights associated with the *SWEBOk Guide* will remain with the IEEE. KA editors must sign a copyright release form.

It is also understood that the *SWEBOk Guide* will continue to be available free of charge in the public domain in at least one format, provided by the IEEE Computer Society through web technology or by other means.

(For more information, see
www.computer.org/copyright.htm.)

REFERENCES

- [1] Project Management Institute, *A Guide to the Project Management Body of Knowledge (PMBOK® Guide)*, 7th ed., Project Management Institute, 2021.
- [2] Integrated Software and Systems Engineering Curriculum (iSSEc) Project, Graduate Software Engineering 2009 (GSwE 2009): Curriculum Guidelines for Graduate Degree Programs in Software Engineering, Stevens Institute of Technology, 2009;

<https://dl.acm.org/doi/book/10.1145/2593248>

[3] ISO/IEC/IEEE 12207-2017 Systems and Software Engineering — Software Life Cycle Processes, 2017.

[4] Joint Task Force on Computing Curricula, IEEE Computer Society and Association for Computing Machinery, “Software Engineering 2014: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering, 2015”;
<https://www.acm.org/binaries/content/assets/education/se2014.pdf>.

[5] ISO/IEC/IEEE 24765:2017 Systems and Software Engineering — Vocabulary, 2nd Edition, ISO/ IEC/IEEE, 2017.

[6] *Merriam-Webster's Collegiate Dictionary*, 11th ed., 2003.

[7] IEEE Computer Society, “Certification and Training for Software Professionals,” 2013;
<https://www.computer.org/education/certifications>.



Home (/) / Volunteering (/volunteering) / Boards and Committees (/volunteering/boards-and-committees) / Professional Educational Activities (/volunteering/boards-and-committees/professional-educational-activities) / Software Engineering Committee (/volunteering/boards-and-committees/professional-educational-activities/software-engineering-committee)

SWEBOK Evolution

IEEE-CS SWEBOK V4 Public Review (3rd Batch)

The IEEE Computer Society Professional & Educational Activities Board (PEAB) SWEBOK Evolution Team seeks public review comments for Version 4 of the Guide to the Software Engineering Body of Knowledge (SWEBOK). This Guide spells out components of the software engineering discipline, promoting a consistent view of software engineering worldwide.

The newest version of the SWEBOK Guide includes new topic areas, updated topic descriptions, and the retirement of no longer relevant topics. Especially, agile (and DevOps) have been incorporated into many knowledge areas (KAs) since these models have been widely accepted since the last publication of SWEBOK. Three new knowledge areas (i.e., Software Architecture, Software Engineering Operations, and Software Security) guide foundational knowledge in software engineering. The new Guide will better integrate the related disciplines and rename and distribute some material into different knowledge areas. V4's table of contents is shown in the following figure.

Software practitioners worldwide participate in the Guide's development to ensure that it captures established traditional practices recommended by many organizations. The SWEBOK Guide uses a rigorous process that includes successive levels of review. Each of the seventeen KAs will be published as they become available for review.

Drafts of the following knowledge areas are now available for review: Introduction, Software Operations KA, Software Maintenance KA, Software Configuration Management KA, Software Engineering Process KA, Software Engineering Models and Methods KA, Computing Foundations KA, and Appendix A Knowledge Area Description Specifications. Please review the drafts, and input your comments in the following form by December 13th, 2022. The input data will be used for SWEBOK V4 editing purposes only. If you have more than five comments, please use the following form multiple times. Please note that all chapters will be finally edited and layout consistently and adequately.

Please let us know if you have questions about this review procedure at swebok-contact@list.waseda.jp (<mailto:swebok-contact@list.waseda.jp>) (Hironori Washizaki, IEEE Computer Society 1st Vice President 2023, SWEBOK V4 Evolution Team Chair).

SWEBOK V4 Public Review Form (3rd batch): <https://forms.gle/9exCgVSFh89esGTh7>
<https://forms.gle/9exCgVSFh89esGTh7>

SWEBOK V4 Drafts in PDF: <https://waseda.box.com/v/ieee-cs-swebok>
<https://waseda.box.com/v/ieee-cs-swebok>

Reference: SWEBOK V3 <https://www.computer.org/education/bodies-of-knowledge/software-engineering> (<https://www.computer.org/education/bodies-of-knowledge/software-engineering>) and http://swebokwiki.org/Main_Page (http://swebokwiki.org/Main_Page)

IEEE-CS Professional & Educational Activities Board (PEAB) SWEBOK Evolution Team
(Vice President for PEAB and Team Chair: Hironori Washizaki)

SWEBOK V3	SWEBOK V4	Public review
Introduction	Introduction	Open
1. Software Requirements	1. Software Requirements	closed
	2. Software Architecture	closed
2. Software Design	3. Software Design	closed
3. Software Construction	4. Software Construction	closed
4. Software Testing	5. Software Testing	closed
	6. Software Engineering Operations	Open
5. Software Maintenance	7. Software Maintenance	Open
6. Software Configuration Management	8. Software Configuration Management	Open
7. Software Engineering Management	9. Software Engineering Management	closed
8. Software Engineering Process	10. Software Engineering Process	Open
9. Software Engineering Models and Methods	11. Software Engineering Models and Methods	Open
10. Software Quality	12. Software Quality	closed
	13. Software Security	closed
11. Software Engineering Professional Practice	14. Software Engineering Professional Practice	closed
12. Software Engineering Economics	15. Software Engineering Economics	closed
13. Computing Foundations	16. Computing Foundations	Open
14. Mathematical Foundations	17. Mathematical Foundations	
15. Engineering Foundations	18. Engineering Foundations	
Appendix A. Knowledge Area Specifications	Appendix A. Knowledge Area Specifications	Open
Appendix B. Standards	Appendix B. Standards	



Sign up for our newsletter.

EMAIL ADDRESS



(<https://www.facebook.com/computersociety/>) (<https://www.linkedin.com/company/ieee-computer-society/>)



IEEE COMPUTER SOCIETY

About Us (<https://www.computer.org/about>)

Board of Governors (/volunteering/board-of-governors)

Newsletters (<https://www.computer.org/resources/newsletters>)

Press Room (<https://www.computer.org/press-room>)

IEEE Support Center (<https://supportcenter.ieee.org/>)

Contact Us (<https://www.computer.org/about/contact>)

DIGITAL LIBRARY

Magazines (<https://www.computer.org/csdl/magazines>)

Journals (<https://www.computer.org/csdl/journals>)

Conference Proceedings (<https://www.computer.org/csdl/proceedings>)

Video Library (<https://www.computer.org/csdl/video-library>)

Librarian Resources (<https://www.computer.org/digital-library/librarian-resources>)

COMPUTING RESOURCES

Jobs Board (<https://jobs.computer.org/>)

Courses & Certifications (<https://www.computer.org/education>)

Webinars (<https://www.computer.org/video-library>)

Podcasts (<https://www.computer.org/resources/podcasts>)

Tech News (<https://www.computer.org/publications/tech-news>)

Membership (<https://www.computer.org/membership/>)

COMMUNITY RESOURCES

Governance (<https://www.computer.org/volunteering/boards-and-committees/resources>)

Conference Organizers (<https://www.computer.org/conferences/organize-a-conference/organizer-resources>)

Authors (<https://www.computer.org/publications/author-resources>)

Chapters (<https://www.computer.org/communities/professional-chapters/resources>)

Communities (<https://www.computer.org/communities>)

BUSINESS SOLUTIONS

Corporate Partnerships (<https://www.computer.org/corporate-programs>)

Conference Sponsorships & Exhibits (<https://www.computer.org/advertising-and-sponsorship-opportunities/conference-sponsorship-and-exhibit-sales?source=advertise>)

Advertising (<https://www.computer.org/advertising-and-sponsorship-opportunities>)

Recruiting (<https://www.computer.org/advertising/recruitment>)

Digital Library Institution Subscriptions (<https://www.computer.org/digital-library/institutional-subscriptions>)

POLICIES

Privacy (<https://www.ieee.org/security-privacy.html>)

Accessibility Statement (<https://www.ieee.org/accessibility-statement.html>)

IEEE Nondiscrimination Policy (<https://www.ieee.org/nondiscrimination>)

XML Sitemap (<https://www.computer.org/sitemap.xml>)

©IEEE — All rights reserved. Use of this website signifies your agreement to the IEEE Terms and Conditions.

A not-for-profit organization, the Institute of Electrical and Electronics Engineers (IEEE) is the world's largest technical professional organization dedicated to advancing technology for the benefit of humanity.