CHAPTER 10

SOFTWARE QUALITY

ACRONYMS

CMMI	Capability Maturity Model Integration
CoSQ	Cost of Software Quality
COTS	Commercial Off-the-Shelf Software
FMEA	Failure Mode and Effects Analysis
FTA	Fault Tree Analysis
PDCA	Plan-Do-Check-Act
PDSA	Plan-Do-Study-Act
QFD	Quality Function Deployment
SPI	Software Process Improvement
SQA	Software Quality Assurance
SQC	Software Quality Control
SQM	Software Quality Management
TQM	Total Quality Management
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 desirable characteristics of software products, to the extent to which a particular software product possess those characteristics, and to processes, tools, and techniques used to achieve those characteristics. Over the years, authors and organizations have defined the term quality differently. To Phil Crosby, it was "conformance to requirements" [1]. Watts Humphrey refers to it as "achieving excellent levels of "fitness for use" [2]. Meanwhile, IBM coined the phrase "market-driven quality," where the "customer is the final arbiter" [3*, p8].

More recently, software quality is defined as the "capability of 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" [5]. Both definitions embrace the premise of conformance to requirements. Neither refers to types of requirements (e.g., functional, reliability, performance, dependability, or any other characteristic). Significantly, however, these definitions emphasize that quality is dependent upon requirements.

These definitions also illustrate another reason for the prevalence of software quality throughout this *Guide*: a frequent ambiguity of *software quality* versus *software quality requirements* ("the *-ilities*" is a common shorthand). Software quality requirements are actually attributes of (or constraints on) functional requirements (what the system does). Software requirements may also specify resource usage, a communication protocol, or many other characteristics. This KA attempts clarity by using *software quality* in the broadest sense from the definitions above and by using *software quality requirements* as constraints on functional requirements. Software quality is achieved by conformance to all requirements regardless of what characteristic is specified or how requirements are grouped or named.

Software quality is also considered in many of the SWEBOK KAs because it is a basic parameter of a software engineering effort. For all engineered products, the primary goal is delivering maximum stakeholder value, while balancing the constraints of development cost and schedule; this is sometimes characterized as "fitness for

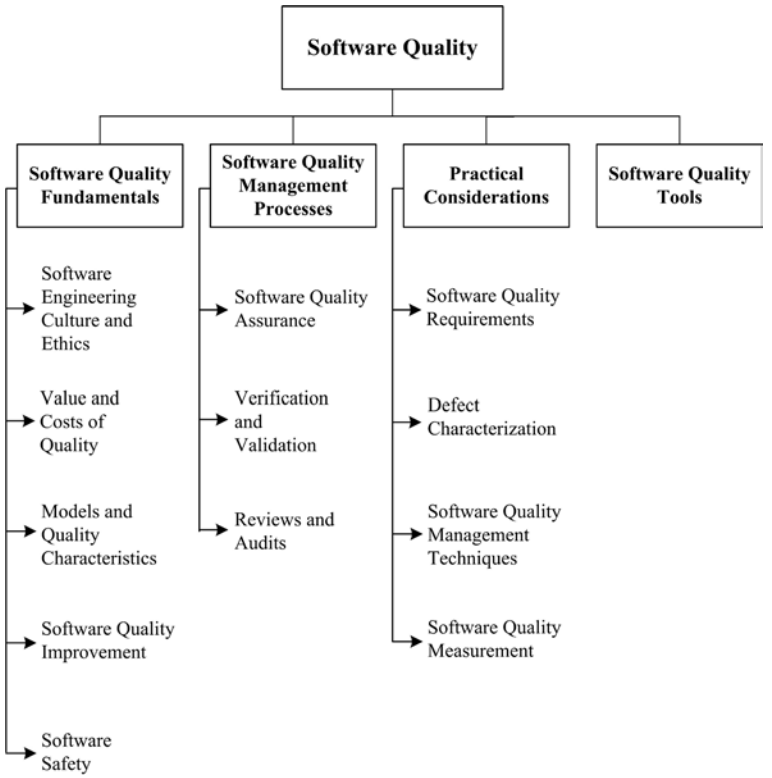

Figure 10.1. Breakdown of Topics for the Software Quality KA

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.

This KA addresses definitions and provides an overview of practices, tools, and techniques for defining software quality and for appraising the state of software quality during development, maintenance, and deployment. Cited references provide additional details.

BREAKDOWN OF TOPICS FOR SOFTWARE QUALITY

The breakdown of topics for the Software Quality KA is presented in Figure 10.1.

1. Software Quality Fundamentals

Reaching agreement on what constitutes quality for all stakeholders and clearly communicating that agreement to software engineers require that the many aspects of quality be formally defined and discussed.

A software engineer should understand quality concepts, characteristics, values, and their application to the software under development or maintenance. The important concept is that the software requirements define the required quality attributes of the software. Software requirements influence the measurement methods and acceptance criteria for assessing the degree to which the software and related documentation achieve the desired quality levels.

1.1. Software Engineering Culture and Ethics

[3*, c1s4] [6*, c2s3.5]

Software engineers are expected to share a commitment to software quality as part of their culture. A healthy software engineering culture includes many characteristics, including the understanding that tradeoffs among cost, schedule, and quality are a basic tenant of the engineering of any product. 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, the culture, and the attitudes of software engineers. The IEEE Computer Society and the 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

[7*, c17, c22]

Defining and then achieving software quality is not simple. Quality characteristics may or may not be required, or they may be required to a greater or lesser degree, and tradeoffs may be made among them. To help determine the level of software quality, i.e., achieving stakeholder value, this section presents cost of software quality (CoSQ): a set of measurements derived from the economic assessment of software quality development and maintenance processes. The CoSQ measurements are examples of process measurements that may be used to infer characteristics of a product.

The premise underlying the CoSQ is that the level of quality in a software product can be inferred from the cost of activities related to dealing with the consequences of poor quality. Poor quality means that the software product does not fully "satisfy stated and implied needs" or "established requirements." There are four cost of quality categories: prevention, appraisal, internal failure, and external failure.

Prevention costs include investments in software process improvement efforts, quality infrastructure, quality tools, training, audits, and management reviews. These costs are usually not specific to a project; they span the organization. Appraisal costs arise from project activities that find defects. These appraisal activities can be categorized into costs of reviews (design, peer) and costs of testing (software unit testing, software integration, system level testing, acceptance testing); appraisal costs would be extended to subcontracted software suppliers. Costs of internal failures are those that are incurred to fix defects found during appraisal activities and discovered prior to delivery of the software product to the customer. External failure costs include activities to respond to software problems discovered after delivery to the customer.

Software engineers should be able to use CoSQ methods to ascertain levels of software quality and should also be able to present quality alternatives and their costs so that tradeoffs between cost, schedule, and delivery of stakeholder value can be made.

1.3. Models and Quality Characteristics

[3*, c24s1] [7*, c2s4] [8*, c17]

Terminology for software quality characteristics differs from one taxonomy (or model of software quality) to another, each model perhaps having a different number of hierarchical levels and a different total number of characteristics. Various authors have produced models of software quality characteristics or attributes that can be useful for discussing, planning, and rating the quality of software products. ISO/IEC 25010: 2011 [4] defines product quality and quality in use as two related quality models. Appendix B in the *SWEBOK Guide* provides a list of applicable standards for each KA. Standards for this KA cover various ways of characterizing software quality.

1.3.1. Software Process Quality

Software quality management and software engineering process quality have a direct bearing on the quality of the software product.

Models and criteria that evaluate 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.

It is not possible to completely distinguish process quality from product quality because process outcomes include products. Determining whether a process has the capability to consistently produce products of desired quality is not simple.

The software engineering process, discussed in the Software Engineering Process KA, influences the quality characteristics of software products, which in turn affect quality as perceived by stakeholders.

1.3.2. Software Product Quality

The software engineer, first of all, must determine the real purpose of the software. In this regard, stakeholder requirements are paramount, and they include quality requirements in addition to functional requirements. Thus, software engineers have a responsibility to elicit quality requirements that may not be explicit at the outset and to understand their importance as well as the level of difficulty in attaining them. All software development processes (e.g., eliciting requirements, designing, constructing, building, checking, improving quality) are designed with these quality requirements in mind and may carry additional development costs if attributes such as safety, security, and dependability are important. The additional development costs help ensure that quality obtained can be traded off against the anticipated benefits.

The term work-product means any artifact that is the outcome of a process used to create the final software product. Examples of a work-product include a system/subsystem specification, a software requirements specification for a software component of a system, a software design description, source code, software test documentation, or reports. While some treatments of quality are described in terms of final software and system performance, sound engineering practice requires that intermediate work-products relevant to quality be evaluated throughout the software engineering process.

1.4. Software Quality Improvement

[3*, c1s4] [9*, c24] [10*, c11s2.4]

The quality of software products can be improved through preventative processes or an iterative process of continual improvement, which requires management control, coordination, and feedback from many concurrent processes: (1) the software life cycle processes, (2) the process of fault/defect detection, removal, and prevention, and (3) the quality improvement process.

The theory and concepts behind quality improvement—such as building in quality through the prevention and early detection of defects, continual improvement, and stakeholder focus—are pertinent to software engineering. These concepts are based on the work of experts in quality who have stated that the quality of a product is directly linked to the quality of the process used to create it. Approaches such as the Deming improvement cycle of Plan-Do-Check-Act (PDCA), evolutionary delivery, kaizen, and quality function deployment (QFD) offer techniques to specify quality objectives and determine whether they are met. The Software Engineering Institute's IDEAL is another method [7*]. Quality management is now recognized by the *SWEBOK Guide* as an important discipline.

Management sponsorship supports process and product evaluations and the resulting findings. Then an improvement program is developed identifying detailed actions and improvement projects to be addressed in a feasible time frame. Management support implies that each improvement project has enough resources to achieve the goal defined for it. Management sponsorship is solicited frequently by implementing proactive communication activities.

1.5. Software Safety

[9*, c11s3]

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 safety-critical. There are increasing numbers of applications of safety-critical software in a growing number of industries. Examples of systems with safety-critical software include mass transit systems, chemical manufacturing plants, and medical devices. The failure of software in these systems could have catastrophic effects. There are industry standards, such as DO-178C [11], and emerging processes, tools, and techniques for developing safetycritical software. The intent of these standards, tools, and techniques is to reduce the risk of injecting faults into the software and thus improve software reliability.

Safety-critical software can be categorized as direct or indirect. Direct is that software embedded in a safety-critical system, such as the flight control computer of an aircraft. Indirect 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 the risk of failure are avoidance, detection and removal, and damage limitation. These techniques impact software functional requirements, software performance requirements, and development processes. Increasing levels of risk imply increasing levels of software quality assurance and control techniques such as inspections. Higher risk levels may necessitate more thorough inspections of requirements, design, and code or the use of more formal analytical 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 and their 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 [12].

2. Software Quality Management Processes

Software quality management is the collection of all processes that ensure that software products, services, and life cycle process implementations meet organizational software quality objectives and achieve stakeholder satisfaction [13, 14]. SQM defines processes, process owners, requirements for the processes, measurements of the processes and their outputs, and feedback channels throughout the whole software life cycle.

SQM comprises four subcategories: software quality planning, software quality assurance (SQA), software quality control (SQC), and software process improvement (SPI). Software quality planning includes determining which quality standards are to be used, defining specific quality goals, and estimating the effort and schedule of software quality activities. In some cases, software quality planning also includes defining the software quality processes to be used. SQA activities 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 [5]. SQC activities examine specific project artifacts (documents and executables) to determine whether they comply with standards established for the project (including requirements, constraints, designs, contracts, and plans). SQC evaluates intermediate products as well as the final products.

The fourth SQM category dealing with improvement has various names within the software industry, including SPI, software quality improvement, and software corrective and preventive action. The activities in this category seek to improve process effectiveness, efficiency, and other characteristics with the ultimate goal of improving software quality. Although SPI could be included in any of the first three categories, an increasing number of organizations organize SPI into a separate category that may span across many projects (see the Software Engineering Process KA).

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 being implemented and how well the intermediate and final products are meeting their specified requirements. Results from these tasks are assembled in reports for management before corrective action is taken. The management of an SQM process is tasked with ensuring that the results of these reports are accurate.

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 Assurance

[7*, c4–c6, c11, c12, c26–27]

To quell a widespread misunderstanding, software quality assurance is not testing. software quality assurance (SQA) is 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 and produce software products of suitable quality for their intended purposes. A key attribute of SQA is the objectivity of the SQA function with respect to the project. The SQA function may also be organizationally independent of the project; that is, free from technical, managerial, and

financial pressures from the project [5]. 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) defines the activities and tasks employed 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 SQA plan's quality activities and tasks are specified 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 SQA plan should be consistent with the software configuration management plan (see the Software Configuration Management KA). The SQA plan identifies documents, standards, practices, and conventions governing the project and how these items are checked and monitored to ensure adequacy and compliance. The SQA plan 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 SQA plan addresses the software quality assurance 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 software (COTS) installation, and service after delivery of the software. It can also contain acceptance criteria as well as reporting and management activities that are critical to software quality.

2.2. Verification & Validation

[9*, c2s2.3, c8, c15s1.1, c21s3.3]

As stated in [15],

> The purpose of V&V is to help the development organization build quality into the system during the life cycle. V&V processes provide an objective assessment of products and processes throughout the life cycle. This assessment demonstrates whether the requirements are correct, complete, accurate, consistent, and testable. The V&V processes determine whether the development products of a given activity conform to the requirements of that activity and whether the product satisfies its intended use and user needs.

Verification is an attempt to ensure that the product is built correctly, in the sense that the output products of an activity meet the specifications imposed on them in previous activities. Validation is an attempt to ensure that the right product is built—that is, the product fulfills its specific intended purpose. Both the verification process and the validation process begin early in the development or maintenance phase. They provide an examination of key product features in relation to both the product's immediate predecessor and the specifications to be met.

The purpose of planning V&V is to ensure that each resource, role, and responsibility is clearly assigned. The resulting V&V plan documents describe the various resources and their roles and activities, as well as the techniques and tools to be used. An understanding of the different purposes of each V&V activity helps in the careful planning of the techniques and resources needed to fulfill their purposes. The plan also addresses the management, communication, policies, and procedures of the V&V activities and their interaction, as well as defect reporting and documentation requirements.

2.3. Reviews and Audits

[9*, c24s3] [16*]

Reviews and audit processes are broadly defined as static—meaning that no software programs or models are executed—examination of software engineering artifacts with respect to standards that have been established by the organization or project for those artifacts. Different types of reviews and audits are distinguished by their purpose, levels of independence, tools and techniques, roles, and by the subject of the activity. Product assurance and process assurance audits are typically conducted by software quality assurance (SQA) personnel who are independent of development

teams. Management reviews are conducted by organizational or project management. The engineering staff conducts technical reviews.

- Management reviews evaluate actual project results with respect to plans.
- Technical reviews (including inspections, walkthrough, and desk checking) examine engineering work-products.
- Process assurance audits. SQA process assurance activities make certain that the processes used to develop, install, operate, and maintain software conform to contracts, comply with any imposed laws, rules, and regulations and are adequate, efficient and effective for their intended purpose [5].
- Product assurance audits. SQA product assurance activities make certain to provide evidence that software products and related documentation are identified in and comply with contracts; and ensure that nonconformances are identified and addressed [5].

2.3.1. Management Reviews

As stated in [16*],

> The purpose of a management review is to monitor progress, determine the status of plans and schedules, and evaluate the effectiveness of management processes, tools and techniques. Management reviews compare actual project results against plans to determine the status of projects or maintenance efforts. The main parameters of management reviews are project cost, schedule, scope, and quality. Management reviews evaluate decisions about corrective actions, changes in the allocation of resources, or changes to the scope of the project.

Inputs to management reviews may include audit reports, progress reports, V&V reports, and plans of many types, including risk management, project management, software configuration management, software safety, and risk assessment, among others. (Refer to the Software Engineering Management and the Software Configuration Management KAs for related material.)

2.3.2. Technical Reviews

As stated in [16*],

> The purpose of a technical review is to evaluate a software product by a team of qualified personnel to determine its suitability for its intended use and identify discrepancies from specifications and standards. It provides management with evidence to confirm the technical status of the project.

Although any work-product can be reviewed, technical reviews are performed on the main software engineering work-products of software requirements and software design.

Purpose, roles, activities, and most importantly the level of formality distinguish different types of technical reviews. Inspections are the most formal, walkthroughs less, and pair reviews or desk checks are the least formal.

Examples of specific roles include a decision maker (i.e., software lead), a review leader, a recorder, and checkers (technical staff members who examine the work-products). Reviews are also distinguished by whether meetings (face to face or electronic) are included in the process. In some review methods checkers solitarily examine work-products and send their results back to a coordinator. In other methods checkers work cooperatively in meetings. A technical review may require that mandatory inputs be in place in order to proceed:

- Statement of objectives
- Specific software product
- Specific project management plan
- Issues list associated with this product
- Technical review procedure.

The team follows the documented review procedure. The technical review is completed once all the activities listed in the examination have been completed.

Technical reviews of source code may include a wide variety of concerns such as analysis of algorithms, utilization of critical computer resources, adherence to coding standards, structure and

organization of code for testability, and safety-critical considerations.

Note that technical reviews of source code or design models such as UML are also termed static analysis (see topic 3, Practical Considerations).

2.3.3. Inspections

"The purpose of an inspection is to detect and identify software product anomalies" [16*]. Some important differentiators of inspections as compared to other types of technical reviews are these:

1. Rules. Inspections are based upon examining a work-product with respect to a defined set of criteria specified by the organization. Sets of rules can be defined for different types of workproducts (e.g., rules for requirements, architecture descriptions, source code).
2. Sampling. Rather that attempt to examine every word and figure in a document, the inspection process allows checkers to evaluate defined subsets (samples) of the documents under review.
3. Peer. Individuals holding management positions over members of the inspection team do not participate in the inspection. This is a key distinction between peer review and management review.
4. Led. An impartial moderator who is trained in inspection techniques leads inspection meetings.
5. Meeting. The inspection process includes meetings (face to face or electronic) conducted by a moderator according to a formal procedure in which inspection team members report the anomalies they have found and other issues.

Software inspections always involve the author of an intermediate or final product; other reviews might not. Inspections also include an inspection leader, a recorder, a reader, and a few (two to five) checkers (inspectors). The members of an inspection team may possess different expertise, such as domain expertise, software design method expertise, or programming language expertise. Inspections are usually conducted on one relatively small section of the product at a time (samples). Each team member examines the software product and other review inputs prior to the review meeting, perhaps by applying an analytical technique (see section 3.3.3) to a small section of the product or to the entire product with a focus on only one aspect—e.g., interfaces. During the inspection, the moderator conducts the session and verifies that everyone has prepared for the inspection and conducts the session. The inspection recorder documents anomalies found. A set of rules, with criteria and questions germane to the issues of interest, is a common tool used in inspections. The resulting list often classifies the anomalies (see section 3.2, Defect Characterization) and is reviewed for completeness and accuracy by the team. The inspection exit decision corresponds to one of the following options:

1. Accept with no or, at most, minor reworking
2. Accept with rework verification
3. Reinspect.

2.3.4. Walkthroughs

As stated in [16*],

> The purpose of a systematic walk-through is to evaluate a software product. A walk-through may be conducted for the purpose of educating an audience regarding a software product.

Walkthroughs are distinguished from inspections. The main difference is that the author presents the work-product to the other participants in a meeting (face to face or electronic). Unlike an inspection, the meeting participants may not have necessarily seen the material prior to the meeting. The meetings may be conducted less formally. The author takes the role of explaining and showing the material to participants and solicits feedback. Like inspections, walkthroughs may be conducted on any type of work-product including project plan, requirements, design, source code, and test reports.

2.3.5. Process Assurance and Product Assurance Audits

As stated in [16*],

> The purpose of a software audit is to provide an independent evaluation of the conformance of software products and processes to applicable regulations, standards, guidelines, plans, and procedures.

Process assurance audits determine the adequacy of plans, schedules, and requirements to achieve project objectives [5]. The audit is a formally organized activity with participants having specific roles—such as lead auditor, another auditor, a recorder, or an initiator—and including a representative of the audited organization. Audits identify instances of nonconformance and produce a report requiring the team to take corrective action.

While there may be many formal names for reviews and audits, such as those identified in the standard [16*], the important point is that they can occur on almost any product at any stage of the development or maintenance process.

3. Practical Considerations

3.1. Software Quality Requirements

[9*, c11s1] [18*, c12] [17*, c15s3.2.2, c15s3.3.1, c16s9.10]

3.1.1. Influence Factors

Various factors influence planning, management, and selection of SQM activities and techniques, including

- the domain of the system in which the software resides; the system functions could be safety-critical, mission-critical, business-critical, security-critical
- the physical environment in which the software system resides
- system and software functional (what the system does) and quality (how well the system performs its functions) requirements
- the commercial (external) or standard (internal) components to be used in the system
- the specific software engineering standards applicable
- the methods and software tools to be used for development and maintenance and for quality evaluation and improvement
- the budget, staff, project organization, plans, and scheduling of all processes
- the intended users and use of the system
- the integrity level of the system.

Information on these factors influences how the SQM processes are organized and documented, how specific SQM activities are selected, what resources are needed, and which of those resources impose bounds on the efforts.

3.1.2. Dependability

In cases where system failure may have extremely severe consequences, overall dependability (hardware, software, and human or operational) is the main quality requirement over and above basic functionality. This is the case for the following reasons: system failures affect a large number of people; users often reject systems that are unreliable, unsafe, or insecure; system failure costs may be enormous; and undependable systems may cause information loss. System and software dependability include such characteristics as availability, reliability, safety, and security. When developing dependable software, tools and techniques can be applied to reduce the risk of injecting faults into the intermediate deliverables or the final software product. Verification, validation, and testing processes, techniques, methods, and tools identify faults that impact dependability as early as possible in the life cycle. Additionally, mechanisms may need to be in place in the software to guard against external attacks and to tolerate faults.

3.1.3. Integrity Levels of Software

Defining integrity levels is a method of risk management.

> Software integrity levels are a range of values that represent software complexity, criticality, risk, safety level, security level,

> desired performance, reliability, or other project-unique characteristics that define the importance of the software to the user and acquirer. 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 to be determined as a part of that system.

The assigned software integrity levels may 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 a tool used in determining software integrity levels. [5]

As noted in [17*], "the integrity levels can be applied during development to allocate additional verification and validation efforts to high-integrity components."

3.2. Defect Characterization

[3*, c3s3, c8s8, c10s2]

Software quality evaluation (i.e., software quality control) techniques find defects, faults and failures. Characterizing these techniques leads to an understanding of the product, facilitates corrections to the process or the product, and informs management and other stakeholders of the status of the process or product. Many taxonomies exist and, while attempts have been made to gain consensus, the literature indicates that there are quite a few in use. Defect characterization is also used in audits and reviews, with the review leader often presenting a list of issues provided by team members for consideration at a review meeting.

As new design methods and languages evolve, along with advances in overall software technologies, new classes of defects appear, and a great deal of effort is required to interpret previously defined classes. When tracking defects, the software engineer is interested in not only the number of defects but also the types. Information alone, without some classification, may not be sufficient to identify the underlying causes of the defects. Specific types of problems need to be grouped to identify trends over time. The point is to establish a defect taxonomy that is meaningful to the organization and to software engineers.

Software quality control activities discover information at all stages of software development and maintenance. In some cases, the word *defect* is overloaded to refer to different types of anomalies. However, different engineering cultures and standards may use somewhat different meanings for these terms. The variety of terms prompts this section to provide a widely used set of definitions [19]:

- *Computational Error*: "the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition."
- *Error*: "A human action that produces an incorrect result." A slip or mistake that a person makes. Also called human error.
- *Defect*: 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 caused by a person committing an error.
- *Fault*: A defect in source code. An "incorrect step, process, or data definition in computer program." The encoding of a human error in source code. Fault is the formal name of a bug.
- *Failure*: An "event in which a system or system component does not perform a required function within specified limits." A failure is produced when a fault is encountered by the processor under specified conditions.

Using these definitions three widely used software quality measurements are defect density (number of defects per unit size of documents), fault density (number of faults per 1K lines of code), and failure intensity (failures per use-hour or per test-hour). 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 to assist in decisions on when to stop testing.

One probable action resulting from SQM findings is to remove the defects from the product under examination (e.g., find and fix bugs, create new build). Other activities attempt to eliminate

the causes of the defects—for example, root cause analysis (RCA). RCA activities include analyzing and summarizing the findings to find root causes and using measurement techniques to improve the product and the process as well as to track the defects and their removal. Process improvement is primarily discussed in the Software Engineering Process KA, with the SQM process being a source of information.

Data on inadequacies and defects found by software quality control techniques may be lost unless they are recorded. For some techniques (e.g., technical reviews, audits, inspections), recorders are present to set down such information, along with issues and decisions. When automated tools are used (see topic 4, Software Quality Tools), the tool output may provide the defect information. Reports about defects are provided to the management of the organization.

3.3. Software Quality Management Techniques

[7*, c7s3] [8*, c17] [9*, c12s5, c15s1, p417] [16*]

Software quality control techniques can be categorized in many ways, but a straightforward approach uses just two categories: static and dynamic. Dynamic techniques involve executing the software; static techniques involve analyzing documents and source code but not executing the software.

3.3.1. Static Techniques

Static techniques examine software documentation (including requirements, interface specifications, designs, and models) and software source code without executing the code. There are many tools and techniques for statically examining software work-products (see section 2.3.2). In addition, tools that analyze source code control flow and search for dead code are considered to be static analysis tools because they do not involve executing the software code.

Other, more formal, types of analytical techniques are known as formal methods. They are notably used to verify software requirements and designs. They have mostly been used in the verification of crucial parts of critical systems, such as specific security and safety requirements. (See also Formal Methods in the Software Engineering Models and Methods KA.)

3.3.2. Dynamic Techniques

Dynamic techniques involve executing the software code. Different kinds of dynamic techniques are performed throughout the development and maintenance of software. Generally, these are testing techniques, but techniques such as simulation and model analysis may be considered dynamic (see the Software Engineering Models and Methods KA). Code reading is considered a static technique, but experienced software engineers may execute the code as they read through it. Code reading may utilize dynamic techniques. This discrepancy in categorizing indicates that people with different roles and experience in the organization may consider and apply these techniques differently.

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.

3.3.3. Testing

Two types of testing may fall under V&V because of their responsibility for the quality of the materials used in the project:

- Evaluation and tests of tools to be used on the project
- Conformance tests (or review of conformance tests) of components and COTS products to be used in the product.

Sometimes an independent (third-party or IV&V) organization may be tasked to perform testing or to monitor the test process V&V may be called upon to evaluate the testing itself: adequacy of plans, processes, and procedures, and adequacy and accuracy of results.

The third party is not the developer, nor is it associated with the development of the product. Instead, the third party is an independent facility, usually accredited by some body of authority. Their purpose is to test a product for conformance to a specific set of requirements (see the Software Testing KA).

3.4. Software Quality Measurement
[3*, c4] [8*, c17] [9*, p90]

Software quality measurements are used to support decision-making. With the increasing sophistication of software, questions of quality go beyond whether or not the software works to how well it achieves measurable quality goals.

Decisions supported by software quality measurement include determining levels of software quality (notably 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; determining when to stop testing and release a product (see Termination under section 5.1, Practical Considerations, in the Software Testing KA); and determining the efficacy of process improvement efforts.

The cost of SQM processes is an issue frequently raised in deciding how a project or a software development and maintenance group should be organized. Often, generic models of cost are used, which 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 the development process. Software quality measurement data collected internally may give a better picture of cost within this project or organization.

While the software quality measurement data may be useful in itself (e.g., the number of defective requirements or the proportion of defective requirements), mathematical and graphical techniques can be applied to aid in the interpretation of the measures (see the Engineering Foundations KA). These techniques include

- descriptive statistics based (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 the list of further readings)
- 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, which the decision makers can use to focus resources and conduct process improvements where they appear to be most needed. Results from trend analysis may indicate that a schedule is being met, such as in testing, or that certain classes of faults may become more likely to occur unless some corrective action is taken in development. The predictive techniques assist in estimating testing effort and schedule and in predicting 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 includes measuring defect occurrences and applying statistical methods to understand the types of defects that occur most frequently. This information may be used by software process improvement for determining methods to prevent, reduce, or eliminate their recurrence. They also aid in understanding trends, how well detection and containment techniques are working, and how well the development and maintenance processes are progressing.

From these measurement methods, defect profiles can be developed 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 expend the effort where problems are most likely to occur. Similarly, benchmarks, or defect counts typical of that domain, may serve as one aid in determining when the product is ready for delivery. Discussion on using data from SQM to improve development and maintenance processes appears in the Software Engineering Management and Software Engineering Process KAs.

4. Software Quality Tools

Software quality tools include 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, in addition to 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 in order to partially automate and control a review process. They allow users to enter defects found during inspections and reviews for later removal.
- Some tools help organizations perform software safety hazard analysis. These tools provide, e.g., automated support for failure mode and effects analysis (FMEA) and fault tree analysis (FTA).
- Tools that support tracking of software problems provide for entry of anomalies discovered during software testing and subsequent analysis, disposition, and resolution. Some tools include support for workflow and for tracking the status of problem resolution.
- Tools that analyze data captured from software engineering environments and software test environments and produce visual displays of quantified data in the form of graphs, charts, and tables. These tools sometimes include the functionality to perform statistical analysis on data sets (for the purpose of discerning trends and making forecasts). Some of these tools provide defect and removal injection rates; defect densities; yields; distribution of defect injection and removal for each of the life cycle phases.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	Kan 2002 [3*]	Bott et al. 2000 [6*]	Galin 2004 [7*]	Naik and Tripathy 2008 [8*]	Sommerville 2011 [9*]	Voland 2003 [10*]	IEEE Std. 1028-2008 [16*]	Moore 2006 [17*]	Wiegers 2003 [18*]
1. Software Quality Fundamentals									
1.1. Software Engineering Culture and Ethics	c1s4	c2s3.5							
1.2. Value and Cost of Quality			c17, c22						
1.3. Models and Quality Characteristics	c24s1		c2s4	c17					
1.4. Software Quality Improvement	c1s4				c24	c11 s2.4			
1.5. Software Safety					c11s3				
2. Software Quality Management Processes									
2.1. Software Quality Assurance			c4–c6, c11, c26–27						
2.2. Verification and Validation					c2 s2.3, c8, c15 s1.1, c21 s3.3				
2.3. Reviews and Audits					c24s3		*		

	Kan 2002 [3*]	Bott et al. 2000 [6*]	Galin 2004 [7*]	Naik and Tripathy 2008 [8*]	Sommerville 2011 [9*]	Voland 2003 [10*]	IEEE Std. 1028-2008 [16*]	Moore 2006 [17*]	Wiegers 2003 [18*]
3. Software Quality Practical Considerations									
3.1. Software Quality Requirements					c11s1			c15 s3.2.2, c15 s3.3.1, c16 s9.10	c12
3.2. Defect Characterization	c3s3, c8s8, c10s2								
3.3. SQM Techniques			c7s3	c17	c12s5, c15s1, p417		*		
3.4. Software Quality Measurement	c4			c17	p90				
4. Software Quality Tools									

FURTHER READINGS

N. Leveson, *Safeware: System Safety and Computers* [20].

This book describes the importance of software safety practices and how these practices can be incorporated into software development projects.

T. Gilb, *Principles of Software Engineering Management* [21].

This is one of the first books on iterative and incremental development techniques. The Evo Method defines quantified goals, frequent time-boxed iterations, measurements of progress toward goals, and adaptation of plans based on actual results.

T. Gilb and D. Graham, *Software Inspection* [22].

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* [23].

This book provides clear, succinct explanations of different peer review methods distinguished by level of formality and effectiveness. Pragmatic guidance for implementing the methods and how to select which methods are appropriate for given circumstances is provided.

N.R. Tague, *The Quality Toolbox*, 2nd ed., [24].

Provides a pragmatic how-to explanation of a comprehensive set of methods, tools, and techniques for solving quality improvement problems. Includes the seven basic quality control tools and many others.

IEEE Std. P730-2013 Draft Standard for Software Quality Assurance Processes [5].

This draft standard expands the SQA processes identified in IEEE/ISO/IEC 12207-2008. P730 establishes standards for initiating, planning, controlling, and executing the software quality assurance processes of a software development or maintenance project. Approval of this draft standard is expected in 2014.

REFERENCES

[1] P.B. Crosby, *Quality Is Free*, McGraw-Hill, 1979.

[2] W. Humphrey, *Managing the Software Process*, Addison-Wesley, 1989.

[3*] S.H. Kan, *Metrics and Models in Software Quality Engineering*, 2nd ed., Addison-Wesley, 2002.

[4] *ISO/IEC 25010:2011 Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—Systems and Software Quality Models*, ISO/IEC, 2011.

[5] *IEEE P730™/D8 Draft Standard for Software Quality Assurance Processes*, IEEE, 2012.

[6*] F. Bott et al., *Professional Issues in Software Engineering*, 3rd ed., Taylor & Francis, 2000.

[7*] D. Galin, *Software Quality Assurance: From Theory to Implementation*, Pearson Education Limited, 2004.

[8*] S. Naik and P. Tripathy, *Software Testing and Quality Assurance: Theory and Practice*, Wiley-Spektrum, 2008.

[9*] P. Clements et al., *Documenting Software Architectures: Views and Beyond*, 2nd ed., Pearson Education, 2010.

[10*] G. Voland, *Engineering by Design*, 2nd ed., Prentice Hall, 2003.

[11] *RTCA DO-178C, Software Considerations in Airborne Systems and Equipment Certification*, Radio Technical Commission for Aeronautics, 2011.

[12] *IEEE Std. 15026.1-2011 Trial-Use Standard Adoption of ISO/IEC TR 15026-1:2010 Systems and Software Engineering—Systems and Software Assurance—Part 1: Concepts and Vocabulary*, IEEE, 2011.

[13] *IEEE Std. 12207-2008 (a.k.a. ISO/IEC 12207:2008) Standard for Systems and Software Engineering—Software Life Cycle Processes*, IEEE, 2008.

[14] *ISO 9000:2005 Quality Management Systems—Fundamentals and Vocabulary*, ISO, 2005.

[15] *IEEE Std. 1012-2012 Standard for System and Software Verification and Validation*, IEEE, 2012.

[16*] *IEEE Std. 1028-2008, Software Reviews and Audits*, IEEE, 2008.

[17*] J.W. Moore, *The Road Map to Software Engineering: A Standards-Based Guide*, Wiley-IEEE Computer Society Press, 2006.

[18*] K.E. Wiegers, *Software Requirements*, 2nd ed., Microsoft Press, 2003.

[19] *ISO/IEC/IEEE 24765:2010 Systems and Software Engineering—Vocabulary*, ISO/IEC/IEEE, 2010.

[20] N. Leveson, *Safeware: System Safety and Computers*, Addison-Wesley Professional, 1995.

[21] T. Gilb, *Principles of Software Engineering Management*, Addison-Wesley Professional, 1988.

[22] T. Gilb and D. Graham, *Software Inspection*, Addison-Wesley Professional, 1993.

[23] K. Wiegers, *Peer Reviews in Software: A Practical Guide*, Addison-Wesley Professional, 2001.

[24] N.R. Tague, *The Quality Toolbox*, 2nd ed., ASQ Quality Press, 2010.

CHAPTER 11

SOFTWARE ENGINEERING PROFESSIONAL PRACTICE

ACRONYMS

ACM	Association for Computing Machinery
BCS	British Computer Society
CSDA	Certified Software Development Associate
CSDP	Certified Software Development Professional
IEC	International Electrotechnical Commission
IEEE CS	IEEE Computer Society
IFIP	International. Federation for Information Processing
IP	Intellectual Property
ISO	International Organization for Standardization
NDA	Non-Disclosure Agreement
WIPO	World Intellectual Property Organization
WTO	World Trade Organization

INTRODUCTION

The Software Engineering Professional Practice knowledge area (KA) is concerned with the knowledge, skills, and attitudes that software engineers must possess to practice software engineering in a professional, responsible, and ethical manner. Because of the widespread applications of software products in social and personal life, the quality of software products can have profound impact on our personal well-being and societal harmony. Software engineers must handle unique engineering problems, producing software with known characteristics and reliability. This requirement calls for software engineers who possess a proper set of knowledge, skills, training, and experience in professional practice.

The term "professional practice" refers to a way of conducting services so as 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 nontechnical aspects. The concept of professional practice can be viewed as being more applicable within those professions that have a generally accepted body of knowledge; codes of ethics and professional conduct with penalties for violations; accepted processes for accreditation, certification, and licensing; and professional societies to provide and administer all of these. Admission to these professional societies is often predicated on a prescribed combination of education and experience.

A software engineer maintains a professional practice by performing all work in accordance with generally accepted practices, standards, and guidelines notably set forth by the applicable professional society. For example, the Association for Computing Machinery (ACM) and IEEE Computer Society (IEEE CS) have established a Software Engineering Code of Ethics and Professional Practice. Both the British Computer Society (BCS) and the International Federation for Information Processing (IFIP) have established similar professional practice standards. ISO/IEC and IEEE have further provided internationally accepted software engineering standards (see Appendix B of this *Guide*). IEEE CS has established two international certification programs (CSDA, CSDP) and a corresponding *Guide to the Software Engineering Body of Knowledge* (*SWEBOK Guide*). All of these are

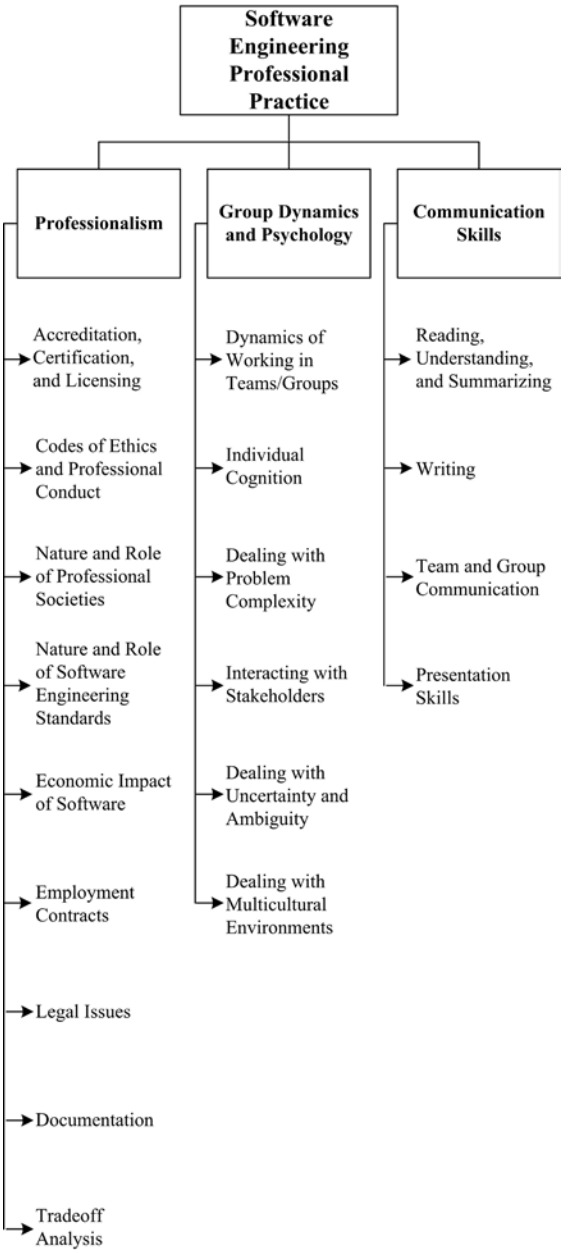

Figure 11.1. Breakdown of Topics for the Software Engineering Professional Practice KA

elements that lay the foundation for of the professional practice of software engineering.

BREAKDOWN OF TOPICS FOR SOFTWARE ENGINEERING PROFESSIONAL PRACTICE

The Software Engineering Professional Practice KA's breakdown of topics is shown in Figure 11.1. The subareas presented in this KA are professionalism, group dynamics and psychology, and communication skills.

1. Professionalism

A software engineer displays professionalism notably through adherence to codes of ethics and professional conduct and to standards and

practices that are established by the engineer's professional community.

The professional community is often represented by one or more professional societies; 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 be used as a measure to determine engineering competence or negligence.

1.1. Accreditation, Certification, and Licensing

[1*, c1s4.1, c1s5.1–c1s5.4]

1.1.1. Accreditation

Accreditation is a process to certify the competency, authority, or credibility of an organization. Accredited schools or programs are assured to adhere to particular standards and maintain certain qualities. In many countries, the basic means by which engineers acquire knowledge is through completion of an accredited course of study. Often, engineering accreditation is performed by a government organization, such as the ministry of education. Such countries with government accreditations include China, France, Germany, Israel, Italy, and Russia.

In other countries, however, the accreditation process is independent of government and performed by private membership associations. For example, in the United States, engineering accreditation is performed by an organization known as ABET. An organization known as CSAB serving as a participating body of ABET is the lead society within ABET for the accreditation of degree programs in software engineering.

While the process of accreditation may be different for each country and jurisdiction, the general meaning is the same. For an institution's course of study to be accredited means that "the accreditation body recognizes an educational institution as maintaining standards that qualify the graduates for admission to higher or more specialized institutions or for professional practice" [2].

1.1.2. Certification

Certification refers to the confirmation of a person's particular characteristics. A common type of certification is professional certification, where a person is certified as being able to complete an activity in a certain discipline at a stated level of competency. Professional certification also can also verify the holder's ability to meet professional standards and to apply professional judgment in solving or addressing problems. Professional certification can also involve the verification of prescribed knowledge, the mastering of best practice and proven methodologies, and the amount of professional experience.

An engineer usually obtains certification by passing an examination in conjunction with other experience-based criteria. These examinations are often administered by nongovernmental organizations, such as professional societies.

In software engineering, certification testifies to one's qualification as a software engineer. For example, the IEEE CS has enacted two certification programs (CSDA and CSDP) designed to confirm a software engineer's knowledge of standard software engineering practices and to advance one's career. A lack of certification does not exclude the individual from working as a software engineer. Currently certification in software engineering is completely voluntary. In fact, most software engineers are not certified under any program.

1.1.3. Licensing

"Licensing" is the action of giving a person the authorization to perform certain kinds of 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 not only that an individual meets a certain standard, but also that they do so with a certain ability to practice or operate. Sometimes there is an entry-level requirement which sets the minimum skills and capabilities to practice, but as the professional moves through his or her career, the required skills and capabilities change and evolve.

In general, engineers are licensed as a means of protecting the public from unqualified individuals. In some countries, no one can practice as a professional engineer unless licensed; or 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*, c1s6–c1s9] [3*, c8] [4*, c1s2] [5*, c33] [6*]

Codes of ethics and professional conduct comprise the values and behavior that an engineer's professional practice and decisions should embody.

The professional community establishes codes of ethics and professional conduct. They exist in the context of, and are adjusted to agree with, societal norms and local laws. Therefore, codes of ethics and professional conduct present guidance in the face of conflicting imperatives.

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 one's abilities. They may also occur through omission, including failure to disclose risks or to provide important information, failure to give proper credit or to acknowledge references, and failure to represent client interests. Violations of codes of ethics and professional conduct may result in penalties and possible expulsion from professional status.

A code of ethics and professional conduct for software engineering was approved by the ACM Council and the IEEE CS Board of Governors in 1999 [6*]. According to the short version of this code:

> Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with 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, respectively.

Since standards and codes of ethics and professional conduct may be introduced, modified, or replaced at any time, individual software engineers bear the responsibility for their own continuing study to stay current in their professional practice.

1.3. Nature and Role of Professional Societies

[1*, c1s1–c1s2] [4*, c1s2] [5*, c35s1]

Professional societies are comprised of a mix of practitioners and academics. These societies serve to define, advance, and regulate their corresponding professions. Professional societies help to establish professional standards as well as codes of ethics and professional conduct. For this reason, they also engage in related activities, which include

- establishing and promulgating a body of generally accepted knowledge;
- accrediting, certifying, and licensing;
- dispensing disciplinary actions;
- advancing the profession through conferences, training, and publications.

Participation in professional societies assists the individual engineer 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*, c5s3.2, c10s2.1] [5*, c32s6] [7*, c1s2]

Software engineering standards cover a remarkable variety of topics. They provide guidelines for the practice of software engineering and processes to be used during development, maintenance, and support of software. By establishing a consensual body of knowledge and experience, software engineering standards establish a basis upon which further guidelines may be developed. Appendix B of this *Guide* provides guidance on IEEE and ISO/IEC software engineering standards that support the knowledge areas of this *Guide*.

The benefits of software engineering standards are many and include improving software quality,

helping avoid errors, protecting both software producers and users, increasing professional discipline, and helping technology transition.

1.5. Economic Impact of Software

[3*, c10s8] [4*, c1s1.1] [8*, c1]

Software has economic effects at the individual, business, and societal levels. Software "success" may be determined by the suitability of a product for a recognized problem as well as 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 or employer's financial situation may in turn be positively or negatively affected by the purchase of software.

At the business level, software properly applied to a problem can eliminate months of work and translate to elevated profits or more effective organizations. Moreover, organizations that acquire or provide successful software may be a boon to the society in which they operate by providing both employment and improved services. However, the development or acquisition costs of software can also equate to those of any major acquisition.

At the societal level, direct impacts of software success or failure include or exclude 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 and life.

1.6. Employment Contracts

[1*, c7]

Software engineering services may be provided under a variety of client-engineer relationships. The software engineering work may be solicited as company-to-customer supplier, engineer-to-customer consultancy, direct hire, or even volunteering. In all of these situations, the customer and supplier agree that a product or service will be provided in return for some sort of consideration. Here, we are most concerned with the engineer-to-customer arrangement and its 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, so they strive to protect that property from disclosure. Therefore, software engineers are often required to sign non-disclosure (NDA) or intellectual property (IP) agreements as a precondition to work. These agreements typically apply to information the software engineer could only gain through association with the customer. The terms of these agreements may extend past termination of the association.

Another concern is IP ownership. Rights to software engineering assets—products, innovations, inventions, discoveries, and ideas—may reside with the employer or customer, either under explicit contract terms or relevant laws, if those assets are obtained during the term of 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 to be performed; standards to which that work will be held; the system configuration to be used for development; limitations of the software engineer's and employer's liability; a communication matrix and/or escalation plan; and administrative details such as rates, frequency of compensation, working hours, and working conditions.

1.7. Legal Issues

[1*, c6, c11] [3*, c5s3–c5s4] [9*, c1s10]

Legal issues surrounding software engineering professional practice notably include matters related to standards, trademarks, patents, copyrights, trade secrets, professional liability, legal requirements, trade compliance, and cybercrime. It is therefore beneficial to possess knowledge of these issues and their applicability.

Legal issues are jurisdictionally based; software engineers must consult attorneys who

specialize in the type and jurisdiction of any identified legal issues.

1.7.1. Standards

Software engineering standards establish guidelines for generally accepted practices and minimum requirements for products and services provided by a software engineer. Appendix B of this *Guide* provides guidance on software engineering standards that are applicable to each KA.

Standards are valuable sources of requirements and assistance during the everyday conduct of software engineering activities. Adherence to standards facilitates discipline by enumerating minimal characteristics of products and practice. That discipline helps to 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. Further, adherence to standards is a major component of defense from legal action or from allegations of malpractice.

1.7.2. Trademarks

A trademark relates to any word, name, symbol, or device that is used in business transactions. It is used "to indicate the source or origin of the goods" [2].

Trademark protection protects names, logos, images, and packaging. However, if a name, image, or other trademarked asset becomes a generic term, then 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 the use of intellectual property 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 a set of exclusive rights granted by a sovereign government to an individual, group of individuals, or organization for a limited period of time. Patents are an old form of idea-ownership protection and date back to the 15th century.

Application for a patent entails careful records of the process that led to the invention. Patent attorneys are helpful in writing patent disclosure claims in a manner most likely to protect the software engineer's rights.

Note that, if inventions are made during the course of a software engineering contract, ownership may belong to the employer or customer or be jointly held, rather than belong to the software engineer.

There are rules concerning what is and is not patentable. In many countries, software code is not patentable, although software algorithms may be. Existing and filed patent applications can be searched at WIPO.

1.7.4. Copyrights

Most governments in the world give exclusive rights of an original work to its creator, usually for a limited time, enacted as a 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 an historical event, whereas the event itself is not protected. Copyrights are long-term and renewable; they date back 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 that these assets are not generally known and may provide a business some economic advantage. The designation of "trade secret" 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 matters of professional liability. As

an individual provides services to a client or employer, it is vital to adhere to standards and generally accepted practices, thereby protecting against allegations or proceedings of or related to malpractice, negligence, or incompetence.

For engineers, including software engineers, professional liability is related to product liability. Under the laws and rules governing in their jurisdiction, engineers may be held to account 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 either 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 (for example, in the US), "privity" (the idea that one could only sue 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. A defense against such an allegation is to show that standards and generally accepted practices were followed in the development of the product.

1.7.7. Legal Requirements

Software engineers must operate within the confines of local, national, and international legal frameworks. Therefore, software engineers must be aware of legal requirements for

- 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 import, export, or reexport of goods, services, and technology in the jurisdictions in which they work. The considerations include export controls and classification, transfer of goods, acquisition of necessary governmental licenses for foreign use of hardware and software, services and technology by sanctioned nation, enterprise 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 it may have been the target. 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 networked 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 separate laws to cover cybercrimes, but it has sometimes been difficult to prosecute cybercrimes due to a lack of precisely framed statutes. The software engineer has a professional obligation to consider the threat of cybercrime and to understand how the software system will protect or endanger software and user information from accidental or malicious access, use, modification, destruction, or disclosure.

1.8. Documentation

[1*, c10s5.8] [3*, c1s5] [5*, c32]

Providing clear, thorough, and accurate documentation is the responsibility of each software engineer. The adequacy of documentation is

judged by different criteria based on the needs of the various stakeholder audiences.

Good documentation complies with accepted standards and guidelines. In particular, software engineers should document

- relevant facts,
- significant risks and tradeoffs, and
- warnings of undesirable or dangerous consequences from use or misuse of the software.

Software engineers should avoid

- certifying or approving unacceptable products,
- disclosing confidential information, or
- falsifying facts or data.

In addition, software engineers and their managers should notably provide the following documentation for use by other elements of the software development organization:

- software requirements specifications, software design documents, details on the software engineering tools used, software test specifications and results, and details on the adopted software engineering methods;
- problems encountered during the development process.

For external stakeholders (customer, users, others) software documentation should notably provide

- information needed to determine if the software is likely to meet the customer's and users' needs,
- description of the safe, and unsafe, use of the software,
- description of the protection of sensitive information created by or stored using the software, and
- clear identification of warnings and critical procedures.

Use of software 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.

The minimum length of time documents should be kept is the duration of the software products' life cycle or the time required by relevant organizational or regulatory requirements.

1.9. Tradeoff Analysis

[3*, c1s2, c10] [9*, c9s5.10]

Within the practice of software engineering, 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 the risks, costs, and benefits of alternatives, in cooperation with stakeholders. The software engineer's evaluation is called "tradeoff analysis." Tradeoff analysis notably enables the identification of competing and complementary software requirements in order to prioritize the final set of 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).

In the case of an ongoing software development project that is late or over budget, tradeoff analysis is often conducted to decide which software requirements can be relaxed or dropped given the effects thereof.

A first step in a tradeoff analysis is establishing design goals (see Engineering Design in the Engineering Foundations KA) and setting the relative importance of those goals. This permits 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 minimization of monetary cost and maximization of reliability, performance, or some other criteria on a wide range of dimensions. However, it is difficult to formulate a tradeoff analysis of cost against risk, especially where primary production and secondary risk-based costs must be traded against each other.

A software engineer must conduct a tradeoff analysis in an ethical manner—notably by being objective and impartial when selecting criteria for comparison of alternative problem solutions and when assigning weights or importance to these criteria. Any conflict of interest must be disclosed up front.

2. Group Dynamics and Psychology

Engineering work is very often conducted in the context of teamwork. A software engineer must be able to interact cooperatively and constructively with others to first determine and then meet both 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] [9*, c1s3.5, c10]

Software engineers must work with others. On 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 that demonstrate 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 that the members naturally commit to and feel ownership of shared outcomes.

Team members facilitate this atmosphere by being intellectually honest, making use of group thinking, admitting ignorance, and acknowledging mistakes. They share responsibility, rewards, and workload fairly. They take care to communicate clearly, directly to each other and in documents, as well as in source code, so that 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 cycle of continuous improvement and growth without personal risk. In general, members of cohesive teams demonstrate respect for each other and their leader.

One point to emphasize is that software engineers must be able to work in multidisciplinary environments and in varied application domains. Since today software is everywhere, from a phone to a car, software is impacting 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 desire to solve problems. The ability to solve problems effectively and efficiently is what every engineer strives for. However, the limits and processes of individual cognition affect problem solving. In software engineering, notably due to the highly abstract nature of software itself, individual cognition plays a very prominent role in problem solving.

In general, an individual's (in particular, a software engineer's) ability to decompose a problem and creatively develop a solution can be inhibited by

- need for more knowledge,
- subconscious assumptions,
- volume of data,
- fear of failure or consequence of failure,
- culture, either application domain or organizational,
- lack of ability to express the problem,
- perceived working atmosphere, and
- emotional status of the individual.

The impact 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 vital, 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.

There is a set of basic methods engineers use 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. Taking advantage of professional curiosity and pursuing continuous professional development

through training and study add skills and knowledge to the software engineer's portfolio; 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] [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, the usual means to adopt is teamwork and problem decomposition (see Problem Solving Techniques in the Computing Foundations KA).

Teams work together to deal with complex and large problems by sharing burdens and drawing upon each other's knowledge and creativity. When software engineers work in teams, different views and abilities of the individual engineers complement each other and help build a solution that is otherwise difficult to come by. Some specific teamwork examples to 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

[9*, c2s3.1]

Success of a software engineering endeavor depends upon positive interactions with stakeholders. They should provide support, information, and feedback at all stages of the software life cycle process. For example, during the early stages, it is critical to identify all stakeholders and discover how the product will affect them, so that sufficient definition of the stakeholder requirements can be properly and completely captured.

During development, stakeholders may provide feedback on specifications and/or early versions of the software, change of priority, as well as clarification of detailed or new software requirements. Last, during software maintenance and until the end of product life, stakeholders provide feedback on evolving or new requirements as well problem reports so that the software may be extended and improved.

Therefore, it is vital to maintain open and productive communication with stakeholders for the duration of the software product's lifetime.

2.5. Dealing with Uncertainty and Ambiguity

[4*, c24s4, c26s2] [9*, c9s4]

As with engineers of other fields, software engineers must often deal with and resolve uncertainty and ambiguities while providing services and developing products. The software engineer must attack and reduce or eliminate any lack of clarity that is an obstacle to performing work.

Often, uncertainty is simply a reflection of lack of knowledge. In this case, investigation through recourse to formal sources such as textbooks and professional journals, interviews with stakeholders, or consultation with teammates and peers can overcome it.

When uncertainty or ambiguity cannot be overcome easily, software engineers or organizations may choose to regard it as a project risk. In this 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 Multicultural Environments

[9*, c10s7]

Multicultural environments can have an impact on the dynamics of a group. This is especially true when the group is geographically separated or communication is infrequent, since such separation elevates the importance of each contact. Intercultural communication is even more difficult if the difference in time zones make oral communication less frequent.

Multicultural environments are quite prevalent in software engineering, perhaps more than in other fields of engineering, due to the strong trend of international outsourcing and the easy shipment of software components instantaneously across the globe. For example, it is rather common for a software project to be divided into pieces across national and cultural borders, and it is also quite common for a software project team to consist of people from diverse cultural backgrounds.

For a software project to be a success, team members must achieve a level of tolerance,

acknowledging that some rules depend on societal norms and that not all societies derive the same solutions and expectations.

This tolerance and accompanying understanding can be facilitated by the support of leadership and management. More frequent communication, including face-to-face meetings, can help to mitigate geographical and cultural divisions, promote cohesiveness, and raise productivity. Also, being able to communicate with teammates in their native language could be very beneficial.

3. Communication Skills

It is vital that a software engineer communicate well, both orally and in reading and writing. Successful attainment of software requirements and deadlines depends on developing clear understanding between the software engineer and 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 usage depend on the provision of relevant training and documentation. It follows that the software engineer's own career success is affected by the ability to consistently provide oral and written communication effectively and on time.

3.1. Reading, Understanding, and Summarizing

[5*, c33s3]

Software engineers are 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 necessary for the completion of engineering goals. A software engineer sifts through accumulated information, filtering out 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 may make the final choice between competing solutions.

Reading and comprehending source code is also a component of information gathering and problem solving. When modifying, extending, or rewriting software, it is critical to understand both its implementation directly derived from the presented code and its design, which must often be inferred.

3.2. Writing

[3*, c1s5]

Software engineers are able to produce written products as required by customer requests or 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.

Writing clearly and concisely is very important because often it is the primary method of communication among relevant parties. In all cases, written software engineering products must be written so that they are accessible, understandable and relevant for their intended audience(s).

3.3. Team and Group Communication

[3*, c1s6.8] [4*, c22s3] [5*, c27s1] [9*, 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.

The number of communication paths, however, grows quadratically with the addition of each team member. Further, 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 sends too many messages, it becomes difficult to identify the important information. Increasingly, organizations are using enterprise

collaboration tools to share information. In addition, the use of electronic information stores, accessible to all team members, for organizational policies, standards, common engineering procedures, and project-specific information, can be most beneficial.

Some software engineering teams focus on face-to-face interaction and promote such interaction by office space arrangement. Although private offices improve individual productivity, colocating team members in physical or virtual forms and providing communal work areas is important to collaborative efforts.

3.4. Presentation Skills

[3*, c1s5] [4*, c22] [9*, c10s7–c10s8]

Software engineers rely on their presentation skills during software life cycle processes. For example, during the software requirements phase, software engineers may walk customers and teammates through software requirements 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. All of these require the ability to present technical information to groups and solicit ideas or feedback.

The software engineer's ability to convey concepts effectively in a presentation therefore 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 the form of slides, knowledge write-up, technical whitepapers, and any other material utilized for knowledge creation.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	Bott et al. 2000 [1*]	Voland 2003 [3*]	Sommerville 2011 [4*]	McConnell 2004 [5*]	IEEE-CS/ACM 1999 [6*]	Moore 2006 [7*]	Tockey 2004 [8*]	Fairley 2009 [9*]
1. Professionalism								
1.1. Accreditation, Certification, and Licensing	c1s4.1, c1s5.1–c1s5.4							
1.2. Codes of Ethics and Professional Conduct	c1s6–c1s9	c8	c1s2	c33	*			
1.3. Nature and Role of Professional Societies	c1s1–c1s2		c1s2	c35s1				
1.4. Nature and Role of Software Engineering Standards	c5s3.2, c10s2.1			c32s6		c1s2		
1.5. Economic Impact of Software		c10s8	c1s1.1				c1	
1.6. Employment Contracts	c7							
1.7. Legal Issues	c6, c11	c5s3–c5s4						c1s10
1.8. Documentation	c10s5.8	c1s5		c32				
1.9. Tradeoff Analysis		c1s2, c10						c9s5.10
2. Group Dynamics and Psychology								
2.1. Dynamics of Working in Teams/ Groups		c1s6						c1s3.5, c10
2.2. Individual Cognition		c1s6.5		c33				
2.3. 2.3 Dealing with Problem Complexity		c3s2		c33				
2.4. Interacting with Stakeholders								c2s3.1

	Bott et al. 2000 [1*]	Voland 2003 [3*]	Sommerville 2011 [4*]	McConnell 2004 [5*]	IEEE-CS/ACM 1999 [6*]	Moore 2006 [7*]	Tockey 2004 [8*]	Fairley 2009 [9*]
2.5. Dealing with Uncertainty and Ambiguity			c24s4, c26s2					c9s4
2.6. Dealing with Multicultural Environments								c10s7
3. Communication Skills								
3.1. Reading, Understanding, and Summarizing				c33s3				
3.2. Writing		c1s5						
3.3. Team and Group Communication		c1s6.8	c22s3	c27s1				c10s4
3.4. Presentation Skills		c1s5	c22					c10s7–c10s8

FURTHER READINGS

Gerald M. Weinberg, *The Psychology of Computer Programming* [10].

This was the first major book to address programming as an individual and team effort and became a classic in the field.

Kinney and Lange, P.A., *Intellectual Property Law for Business Lawyers* [11].

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] *Merriam-Webster's Collegiate Dictionary*, 11th ed., 2003.

[3*] G. Voland, *Engineering by Design*, 2nd ed., Prentice Hall, 2003.

[4*] I. Sommerville, *Software Engineering*, 9th ed., Addison-Wesley, 2011.

[5*] S. McConnell, *Code Complete*, 2nd ed., Microsoft Press, 2004.

[6*] IEEE CS/ACM Joint Task Force on Software Engineering Ethics and Professional Practices, "Software Engineering Code of Ethics and Professional Practice (Version 5.2)," 1999; www.acm.org/serving/se/code.htm.

[7*] J.W. Moore, *The Road Map to Software Engineering: A Standards-Based Guide*, Wiley-IEEE Computer Society Press, 2006.

[8*] S. Tockey, *Return on Software: Maximizing the Return on Your Software Investment*, Addison-Wesley, 2004.

[9*] R.E. Fairley, *Managing and Leading Software Projects*, Wiley-IEEE Computer Society Press, 2009.

[10] G.M. Weinberg, *The Psychology of Computer Programming: Silver Anniversary Edition*, Dorset House, 1998.

[11] Kinney and Lange, P.A., *Intellectual Property Law for Business Lawyers*, Thomson West, 2013.

CHAPTER 12

SOFTWARE ENGINEERING ECONOMICS

ACRONYMS

EVM	Earned Value Management
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
ROCE	Return on Capital Employed
TCO	Total Cost of Ownership

INTRODUCTION

Software engineering economics is about making decisions related to software engineering in a business context. The success of a software product, service, and solution depends on good business management. Yet, in many companies and organizations, software business relationships to software development and engineering remain vague. This knowledge area (KA) provides an overview on software engineering economics.

Economics is the study of value, costs, resources, and their relationship in a given context or situation. In the discipline of software engineering, activities have costs, but the resulting software itself has economic attributes as well. Software engineering economics provides a way to study the attributes of software and software processes in a systematic way that relates them to economic measures. These economic measures can be weighed and analyzed when making decisions that are within the scope of a software organization and those within the integrated scope of an entire producing or acquiring business.

Software engineering economics is concerned with aligning software technical decisions with the business goals of the organization. In all types of organizations—be it "for-profit," "not-for-profit," or governmental—this translates into sustainably staying in business. In "for-profit" organizations this additionally relates to achieving a tangible return on the invested capital—both assets and capital employed. This KA has been formulated in a way to address all types of organizations independent of focus, product and service portfolio, or capital ownership and taxation restrictions.

Decisions like "Should we use a specific component?" may look easy from a technical perspective, but can have serious implications on the business viability of a software project and the resulting product. Often engineers wonder whether such concerns apply at all, as they are "only engineers." Economic analysis and decision-making are important engineering considerations because engineers are capable of evaluating decisions both technically and from a business perspective. The contents of this knowledge area are important topics for software engineers to be aware of even if they are never actually involved in concrete business decisions; they will have a well-rounded view of business issues and the role technical considerations play in making business decisions. Many engineering proposals and decisions, such as make versus buy, have deep intrinsic economic impacts that should be considered explicitly.

This KA first covers the foundations, key terminology, basic concepts, and common practices of software engineering economics to indicate how decision-making in software engineering includes, or should include a business perspective. It then provides a life cycle perspective, highlights risk and uncertainty management, and shows how economic analysis methods are used. Some practical considerations finalize the knowledge area.

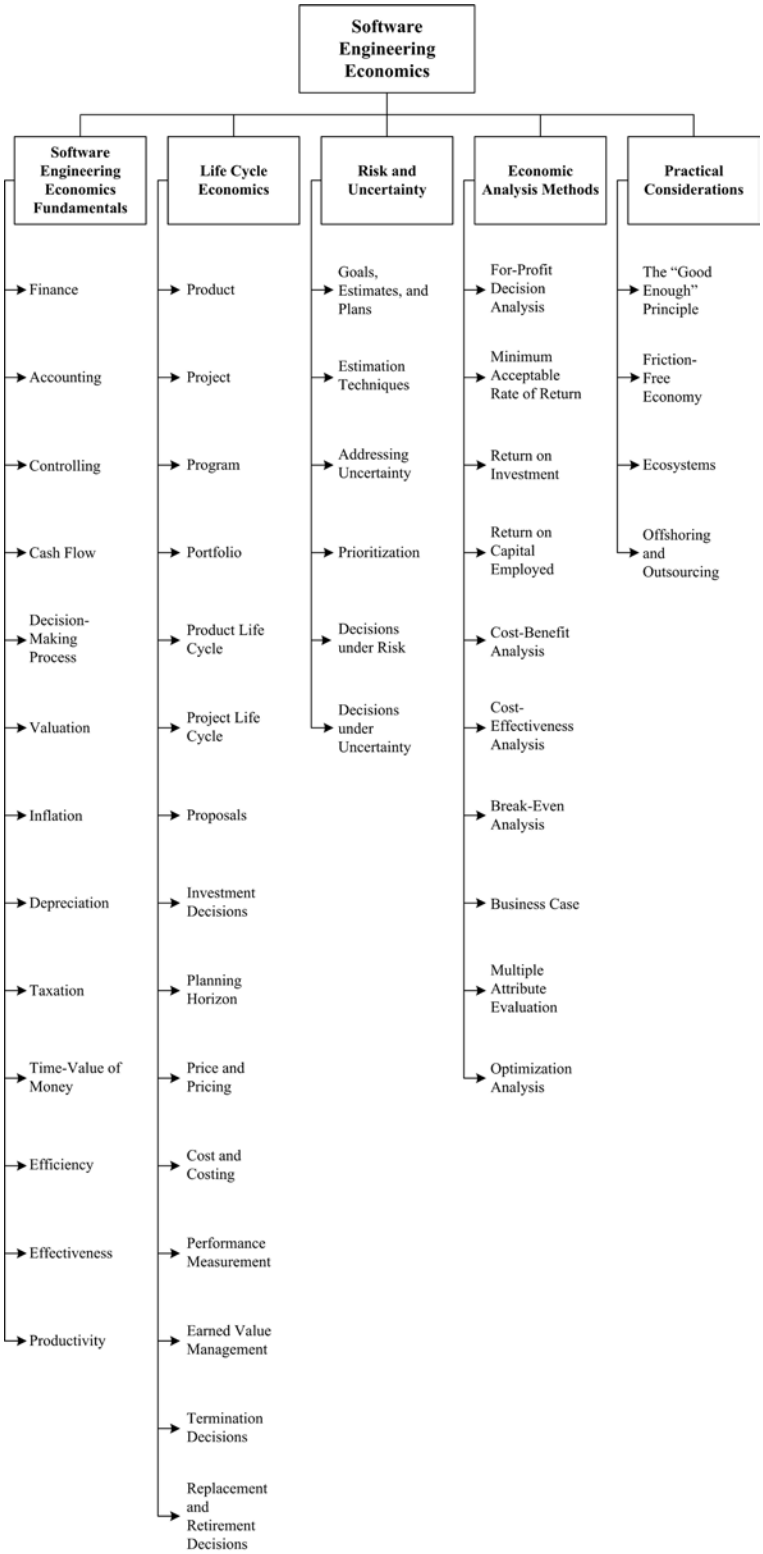

Figure 12.1. Breakdown of Topics for the Software Engineering Economics KA

BREAKDOWN OF TOPICS FOR SOFTWARE ENGINEERING ECONOMICS

The breakdown of topics for the Software Engineering Economics KA is shown in Figure 12.1.

1. Software Engineering Economics Fundamentals

1.1. Finance

[1*, c2]

Finance is the branch of economics concerned with issues such as allocation, management, acquisition, and investment of resources. Finance is an element of every organization, including software engineering organizations.

The field of finance deals with the concepts of time, money, risk, and how they are interrelated. It also deals with how money is spent and budgeted. Corporate finance is concerned with providing the funds for 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 holds primarily for "for-profit" organizations, but also applies to "not-for-profit" organizations. The latter needs finances to ensure sustainability, while not targeting 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 (see section 4.3, Return on Investment), and take corrective actions in case of deviation from objectives and strategy.

1.2. Accounting

[1*, c15]

Accounting is part of finance. It allows people whose money is being 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 ROI (see section 4.3, Return on Investment), while in "not-for-profit" and governmental organizations as well as "for-profit" organizations, it translates into sustainably staying in business. The primary role of accounting 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 is generally in the form of financial statements that show in money terms the economic resources to be controlled. It is important to select the right information that is both relevant and reliable to the user. 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.

1.3. Controlling

[1*, c15]

Controlling is an element of finance and accounting. Controlling involves measuring and correcting the performance of finance and accounting. 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.

1.4. Cash Flow

[1*, c3]

Cash flow is the movement of money into or out of a business, project, or financial product over a given period. The concepts of cash flow instances and cash flow streams are used to describe the business perspective of a proposal. To make a meaningful business decision about any specific proposal, that proposal will need to be evaluated from a business perspective. In a proposal to develop and launch product X, the payment for new software licenses is 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 is an example of an incoming cash flow

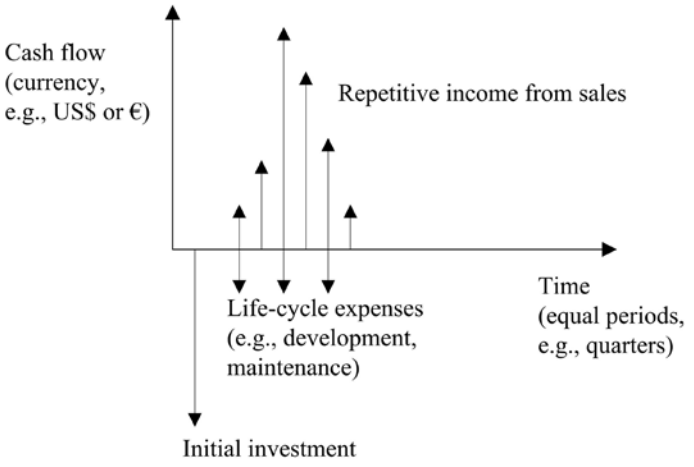

Figure 12.2. A Cash Flow Diagram

instance. Money would be coming in because of carrying out the proposal.

The term *cash flow stream* refers to the set of cash flow instances over time that are caused by carrying out some given proposal. The cash flow stream is, in effect, the complete financial picture of that proposal. How much money goes out? When does it go out? How much money comes in? When does it come in? Simply, 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 better off carrying out Proposal A than Proposal B. Thus, the cash flow stream is an important input for investment decision-making. 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 some activity.

A cash flow diagram is a picture of a cash flow stream. It gives the reader a quick overview of the financial picture of the subject organization or project. Figure 12.2 shows an example of a cash flow diagram for a proposal.

1.5. Decision-Making Process

[1*, c2, c4]

If we assume that candidate solutions solve a given technical problem equally well, why should the organization care which one is chosen? The answer is that there is usually a large difference in the costs and incomes from the different solutions. A commercial, off-the-shelf, object-request broker product might cost a few thousand dollars, but the effort to develop a homegrown service that gives the same functionality could easily cost several hundred times that amount.

If the candidate solutions all adequately solve the problem from a technical perspective, then the selection of the most appropriate alternative should be based on commercial factors such as optimizing total cost of ownership (TCO) or maximizing the short-term return on investment (ROI). Life cycle costs such as defect correction, field service, and support duration are also relevant considerations. These costs need to be factored in when selecting among acceptable technical approaches, as they are part of the lifetime ROI (see section 4.3, Return on Investment).

A systematic process for making decisions will achieve transparency and allow later justification. Governance criteria in many organizations demand selection from at least two alternatives. A systematic process is shown in Figure 12.3. It starts with a business challenge at hand and describes the steps to identify alternative solutions, define selection criteria, evaluate the solutions, implement one selected solution, and monitor the performance of that solution.

Figure 12.3 shows the process as mostly stepwise and serial. The real process is more fluid. Sometimes the steps can be done in a different order and often several of the steps can be done in parallel. The important thing is to be sure that

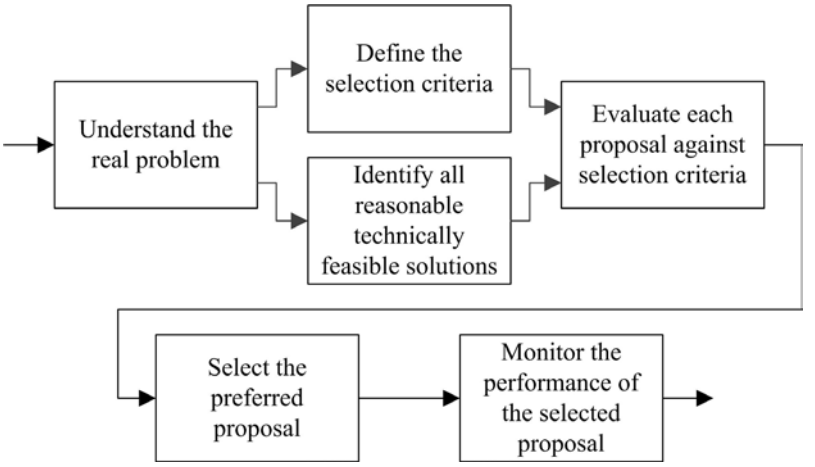

Figure 12.3. The Basic Business Decision-Making Process

none of the steps are skipped or curtailed. It's also important to understand that this same process applies at all levels of decision making: from a decision as big as determining whether a software project should be done at all, to a deciding on an algorithm or data structure to use in a software module. The difference is how financially significant the decision is and, therefore, how much effort should be invested in making that decision. The project-level decision is financially significant and probably warrants a relatively high level of effort to make the decision. Selecting an algorithm is often much less financially significant and warrants a much lower level of effort to make the decision, even though the same basic decision-making process is being used.

More often than not, an organization could carry out more than one proposal if it wanted to, and usually there are important relationships among proposals. 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. Choices are much easier to make when there are mutually exclusive paths—for example, either A or B or C or whatever is chosen. In preparing decisions, it is recommended to turn any given set of proposals, along with their various interrelationships, into a set of mutually exclusive alternatives. The choice can then be made among these alternatives.

1.6. Valuation

[1*, c5, c8]

In an abstract sense, the decision-making process—be it financial decision making or other—is about maximizing value. The alternative that maximizes total value should always be chosen. A financial basis for value-based comparison is comparing two or more cash flows. Several bases of comparison are available, including

- present worth
- future worth
- annual equivalent
- internal rate of return
- (discounted) payback period.

Based on the time-value of money, two or more cash flows are equivalent only when they equal the same amount of money at a common point in time. Comparing cash flows only makes sense when they are expressed in the same time frame.

Note that value can't always be expressed in terms of money. For example, whether an item is a brand name or not can significantly affect its perceived value. Relevant values that can't be expressed in terms of money still need to be expressed in similar terms so that they can be evaluated objectively.

1.7. Inflation

[1*, c13]

Inflation describes long-term trends in prices. Inflation means that the same things cost more than they did before. If the planning horizon of a business decision is longer than a few years, or if the inflation rate is over a couple of percentage points annually, it can cause noticeable changes in the value of a proposal. The present time value therefore needs to be adjusted for inflation rates and also for exchange rate fluctuations.

1.8. Depreciation

[1*, c14]

Depreciation involves spreading the cost of a tangible asset across a number of time periods; it is used to determine how investments in capitalized assets are charged against income over several years. Depreciation is an important part of determining after-tax cash flow, which is critical for accurately addressing profit and taxes. If a software product is to be sold after the development costs are incurred, those costs should be capitalized and depreciated over subsequent time periods. The depreciation expense for each time period is the capitalized cost of developing the software divided across the number of periods in which the software will be sold. A software project proposal may be compared to other software and nonsoftware proposals or to alternative investment options, so it is important to determine how those other proposals would be depreciated and how profits would be estimated.

1.9. Taxation

[1*, c16, c17]

Governments charge taxes in order to finance expenses that society needs but that no single organization would invest in. Companies have to pay income taxes, which can take a substantial portion of a corporation's gross profit. A decision analysis that does not account for taxation can lead to the wrong choice. A proposal with a high pretax profit won't look nearly as profitable in posttax terms. Not accounting for taxation can also lead to unrealistically high expectations about how profitable a proposed product might be.

1.10. Time-Value of Money

[1*, c5, c11]

One of the most fundamental concepts in finance—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 different value than the same amount of money at some other time. This concept has been around since the earliest recorded human history and is commonly known as *time-value*. In order to compare proposals or portfolio elements, they should be normalized in cost, value, and risk to the net present value. Currency exchange variations over time need to be taken into account based on historical data. This is particularly important in cross-border developments of all kinds.

1.11. Efficiency

[2*, c1]

Economic efficiency of a process, activity, or task is the ratio of resources actually consumed to resources expected to be consumed or desired to be consumed in accomplishing the process, activity, or task. Efficiency means "doing things right." An efficient behavior, like an effective behavior, delivers results—but keeps the necessary effort to a minimum. Factors that may affect efficiency in software engineering include product complexity, quality requirements, time pressure, process capability, team distribution, interrupts, feature churn, tools, and programming language.

1.12. Effectiveness

[2*, c1]

Effectiveness is about having impact. It is the relationship between achieved objectives to defined objectives. Effectiveness means "doing the right things." Effectiveness looks only at whether defined objectives are reached—not at how they are reached.

1.13. Productivity

[2*, c23]

Productivity is the ratio of output over 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 highest value with lowest resource consumption.

2. Life Cycle Economics

2.1. Product

[2*, c22] [3*, c6]

A product is an economic good (or output) that is created in a process that transforms product factors (or inputs) to an output. When sold, a product is a deliverable that creates both a value and an experience for its users. A product can be a combination of systems, solutions, materials, and services delivered internally (e.g., in-house IT solution) or externally (e.g., software application), either as-is or as a component for another product (e.g., embedded software).

2.2. Project

[2*, c22] [3*, c1]

A project is "a temporary endeavor undertaken to create a unique product, service, or result".[1] 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 the customer need and market requirements; during maintenance, a project might be conducted to produce a next version of a product.

2.3. 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."[2] Programs are often used to identify and manage different deliveries to a single customer or market over a time horizon of several years.

2.4. Portfolio

Portfolios are "projects, programs, subportfolios, and operations managed as a group to achieve strategic objectives."[3] Portfolios are used to group and then manage simultaneously all assets within a business line or organization. Looking to an entire portfolio makes sure that impacts of decisions are considered, such as resource allocation to a specific project—which means that the same resources are not available for other projects.

2.5. Product Life Cycle

[2*, c2] [3*, c2]

A software product life cycle (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" typically 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 typically 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 the activities after initial product release.

2.6. Project Life Cycle

[2*, c2] [3*, c2]

Project life cycle activities typically involve five process groups—Initiating, Planning, Executing, Monitoring and Controlling, and Closing [4]

1 Project Management Institute, Inc., *PMI Lexicon of Project Management Terms,* 2012, www.pmi.org/PMBOK-Guide-and-Standards/~/media/Registered/PMI_Lexicon_Final.ashx.

2 Ibid.

3 Ibid.

(see the Software Engineering Management KA). The activities within a software project life cycle are often interleaved, overlapped, and iterated in various ways [3*, c2] [5] (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., complying with product liability needs or governance regulations). The software project life cycle and the software product life cycle are interrelated; an SPLC may include several SDLCs.

2.7. Proposals

[1*, c3]

Making a business decision begins with the notion of a *proposal*. Proposals relate to reaching a business objective—at the project, product, or portfolio level. A proposal is a single, separate option that is being considered, like carrying out a particular software development project or not. Another proposal could be to enhance an existing software component, and still another might be to redevelop that same software from scratch. Each proposal represents a unit of choice—either you can choose to carry out that proposal or you can choose not to. The whole purpose of business decision-making is to figure out, given the current business circumstances, which proposals should be carried out and which shouldn't.

2.8. Investment Decisions

[1*, c4]

Investors make investment decisions to spend money and resources on achieving a target objective. Investors are either inside (e.g., finance, board) or outside (e.g., banks) the organization. The target relates to some economic criteria, such as achieving a high return on the investment, strengthening the capabilities of the organization, or improving the value of the company. Intangible aspects such as goodwill, culture, and competences should be considered.

2.9. Planning Horizon

[1*, c11]

When an organization chooses to invest in a particular proposal, money gets tied up in that proposal—so-called "frozen assets." The economic impact of frozen assets tends to start high and decreases over time. On the other hand, operating and maintenance costs of elements associated with the proposal tend to start low but increase over time. The total cost of the proposal—that is, owning and operating a product—is the sum of those two costs. Early on, frozen asset costs dominate; later, the operating and maintenance costs dominate. There is a point in time where the sum of the costs is minimized; this is called the *minimum cost lifetime*.

To properly compare a proposal with a four-year life span to a proposal with a six-year life span, 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 proposals are considered. Effects such as software lifetime will 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.

2.10. Price and Pricing

[1*, c13]

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 amongst the four Ps; the rest are costs.

Pricing is an element of finance and marketing. It is the process of determining what a company will receive in exchange for its products. Pricing factors include manufacturing cost, market placement, competition, market condition, and quality of product. 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 needs of the consumer can be converted into demand only if the consumer has the willingness and capacity to buy the product. Thus, pricing is very important in marketing. Pricing is initially done during the project initiation phase and is a part of "go" decision making.

2.11. Cost and Costing

[1*, c15]

A cost is the value of money that has been used up to produce something and, hence, is not available for use anymore. In economics, a cost is an alternative that is given up as a result of a decision.

A sunk cost is the expenses before a certain time, typically used to abstract decisions from expenses in the past, which can cause emotional hurdles in looking forward. From a traditional economics point of view, sunk costs should not be considered in decision making. Opportunity cost is the cost of an alternative that must be forgone in order to pursue another alternative.

Costing is part of finance and product management. It is the process to determine 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 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, activating, and keeping that product running. These costs can be grouped as direct and indirect costs. TCO is an accounting method that is crucial in making sound economic decisions.

2.12. Performance Measurement

[3*, c7, c8]

Performance measurement is the process whereby an organization establishes and measures the parameters used to determine whether programs, investments, and acquisitions are achieving the desired results. It is used to evaluate whether performance objectives are actually achieved; to control budgets, resources, progress, and decisions; and to improve performance.

2.13. Earned Value Management

[3*, c8]

Earned value management (EVM) is a project management technique for measuring progress based on created value. At a given moment, the results achieved to date in a project are compared with the projected budget and the planned schedule progress for that date. Progress relates already-consumed resources and achieved results at a given point in time with the respective planned values for the same date. It helps to identify possible performance problems at an early stage. A key principle in EVM is tracking cost and schedule variances via comparison of planned versus actual schedule and budget versus actual cost. EVM tracking gives much earlier visibility to deviations and thus permits corrections earlier than classic cost and schedule tracking that only looks at delivered documents and products.

2.14. Termination Decisions

[1*, c11, c12] [2*, c9]

Termination means to end a project or product. Termination can be preplanned for the end of a long product lifetime (e.g., when foreseeing that a product will reach its lifetime) or can come rather spontaneously during product development (e.g., when project performance targets are not achieved). In both cases, the decision should be carefully prepared, considering always the alternatives of continuing versus terminating. Costs of different alternatives must be estimated—covering topics such as replacement, information collection, suppliers, alternatives, assets, and utilizing resources for other opportunities. Sunk costs should not be considered in such decision making because they have been spent and will not reappear as a value.

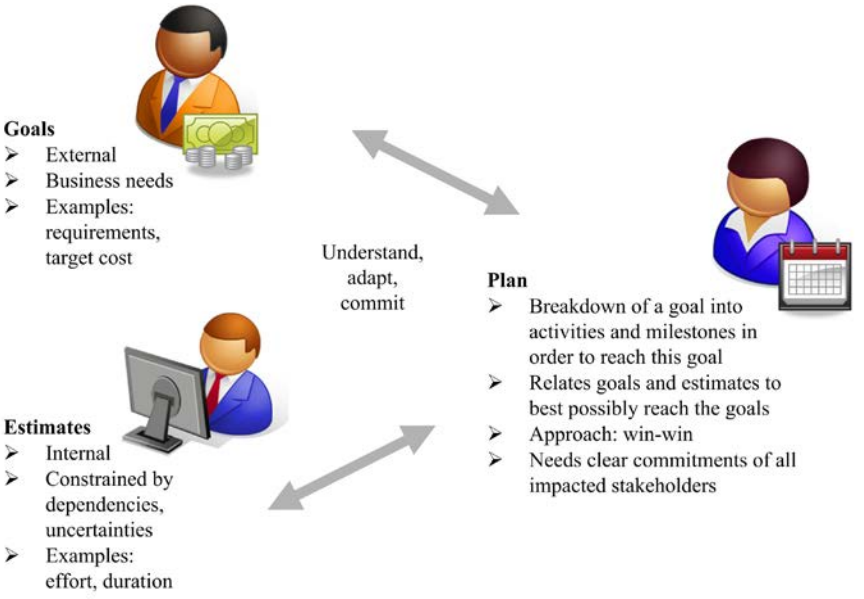

Figure 12.4. Goals, Estimates, and Plans

2.15. Replacement and Retirement Decisions

[1*, c12] [2*, c9]

A replacement decision is made when an organization already has a particular asset and they are considering replacing it with something else; for example, deciding between maintaining and supporting a legacy software product or redeveloping it from the ground up. Replacement decisions use the same business decision process as described above, but there are additional challenges: sunk cost and salvage value. Retirement decisions are also 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 model of computer any longer. Retirement decision can be influenced by lock-in factors such as technology dependency and high exit costs.

3. Risk and Uncertainty

3.1. Goals, Estimates, and Plans

[3*, c6]

Goals in software engineering economics are mostly business goals (or business objectives). A business goal relates business needs (such as increasing profitability) to investing resources (such as starting a project or launching a product with a given budget, content, and timing). Goals apply to operational planning (for instance, to reach a certain milestone at a given date or to extend software testing by some time to achieve a desired quality level—see Key Issues in the Software Testing KA) and to the strategic level (such as reaching a certain profitability or market share in a stated time period).

An estimate is a well-founded evaluation of resources and time that will be needed to achieve stated goals (see Effort, Schedule, and Cost Estimation in the Software Engineering Management KA and Maintenance Cost Estimation in the Software Maintenance KA). A software estimate is used to determine whether the project goals can be achieved within the constraints on schedule, budget, features, and quality attributes. Estimates are typically internally generated and are not necessarily visible externally. Estimates should not be driven exclusively by the project goals because this could make an estimate overly optimistic. Estimation is a periodic activity; estimates should be continually revised during a project.

A plan describes the activities and milestones that are necessary in order to reach the goals of

a project (see Software Project Planning in the Software Engineering Management KA). The plan should be in line with the goal and the estimate, which is not necessarily easy and obvious—such as when a software project with given requirements would take longer than the target date foreseen by the client. In such cases, plans demand a review of initial goals as well as estimates and the underlying uncertainties and inaccuracies. Creative solutions with the underlying rationale of achieving a win-win position are applied to resolve conflicts.

To be of value, planning should involve consideration of the project constraints and commitments to stakeholders. Figure 12.4 shows how goals are initially defined. Estimates are done based on the initial goals. The plan tries to match the goals and the estimates. This is an iterative process, because an initial estimate typically does not meet the initial goals.

3.2. Estimation Techniques

[3*, c6]

Estimations are used to analyze and forecast the resources or time necessary to implement requirements (see Effort, Schedule, and Cost Estimation in the Software Engineering Management KA and Maintenance Cost Estimation in the Software Maintenance KA). Five families of estimation techniques exist:

- Expert judgment
- Analogy
- Estimation by parts
- Parametric methods
- Statistical methods.

No single estimation technique is perfect, so using multiple estimation technique is useful. Convergence among the estimates produced by different techniques indicates that the estimates are probably accurate. Spread among the estimates indicates that certain factors might have been overlooked. Finding the factors that caused the spread and then reestimating again to produce results that converge could lead to a better estimate.

3.3. Addressing Uncertainty

[3*, c6]

Because of the many unknown factors during project initiation and planning, estimates are inherently uncertain; that uncertainty should be addressed in business decisions. Techniques for addressing uncertainty include

- consider ranges of estimates
- analyze sensitivity to changes of assumptions
- delay final decisions.

3.4. Prioritization

[3*, c6]

Prioritization involves ranking alternatives based on common criteria to deliver the best possible value. In software engineering projects, software requirements are often prioritized in order to deliver the most value to the client within constraints of schedule, budget, resources, and technology, or to provide for building product increments, where the first increments provide the highest value to the customer (see Requirements Classification and Requirements Negotiation in the Software Requirements KA and Software Life Cycle Models in the Software Engineering Process KA).

3.5. Decisions under Risk

[1*, c24] [3*, c9]

Decisions under risk techniques are used when the decision maker can assign probabilities to the different possible outcomes (see Risk Management in the Software Engineering Management KA). The specific techniques include

- expected value decision making
- expectation variance and decision making
- Monte Carlo analysis
- decision trees
- expected value of perfect information.

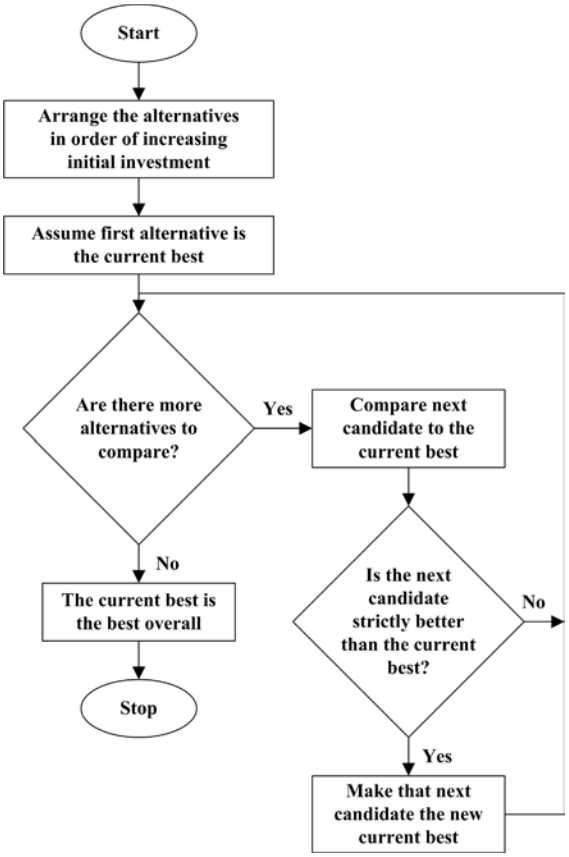

Figure 12.5. The for-profit decision-making process

3.6. Decisions under Uncertainty

[1*, c25] [3*, c9]

Decisions under uncertainty techniques are used when the decision maker cannot assign probabilities to the different possible outcomes because needed information is not available (see Risk Management in the Software Engineering Management KA). Specific techniques include

- Laplace Rule
- Maximin Rule
- Maximax Rule
- Hurwicz Rule
- Minimax Regret Rule.

4. Economic Analysis Methods

4.1. For-Profit Decision Analysis

[1*, c10]

Figure 12.5 describes a process for identifying the best alternative from a set of mutually exclusive alternatives. Decision criteria depend on the business objectives and typically include ROI (see section 4.3, Return on Investment) or Return on Capital Employed (ROCE) (see section 4.4, Return on Capital Employed).

For-profit decision techniques don't apply for government and nonprofit organizations. In these cases, organizations have different goals—which means that a different set of decision techniques are needed, such as cost-benefit or cost-effectiveness analysis.

4.2. Minimum Acceptable Rate of Return

[1*, c10]

The minimum acceptable rate of return (MARR) is the lowest internal rate of return the organization would consider to be a good investment. Generally speaking, it wouldn't be smart to invest in an activity with a return of 10% when there's another activity that's known to return 20%. The MARR is a statement that an organization is confident it can achieve at least that rate of return. The MARR represents the organization's opportunity cost for investments. By choosing to invest in some activity, the organization is explicitly deciding to not invest that same money somewhere else. If the organization is already confident it can get some known rate of return, other 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 business decisions. An alternative's present worth evaluated at the MARR shows how much more or less (in present-day cash terms) that alternative is worth than investing at the MARR.

4.3. Return on Investment

[1*, c10]

Return on investment (ROI) is a measure of the profitability of a company or business unit. It is defined as the ratio of money gained or lost (whether realized or unrealized) on an investment relative to the amount of money invested. The purpose of ROI varies and includes, for instance, providing a rationale for future investments and acquisition decisions.

4.4. Return on Capital Employed

The return on capital employed (ROCE) is a measure of the profitability of a company or business unit. It is defined as the ratio of a gross profit before taxes and interest (EBIT) to the total assets minus current liabilities. It describes the return on the used capital.

4.5. Cost-Benefit Analysis

[1*, c18]

Cost-benefit analysis is one of the most widely used methods for evaluating individual proposals. 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 the benefit. Proposals with a higher ratio need to consider the associated risk of an investment and compare the benefits with the option of investing the money at a guaranteed interest rate (see section 4.2, Minimum Acceptable Rate of Return).

4.6. Cost-Effectiveness Analysis

[1*, c18]

Cost-effectiveness analysis is similar to cost-benefit analysis. There are two versions of cost-effectiveness analysis: the *fixed-cost* version maximizes the benefit given some upper bound on cost; the *fixed-effectiveness* version minimizes the cost needed to achieve a fixed goal.

4.7. Break-Even Analysis

[1*, c19]

Break-even analysis identifies the point where the costs of developing a product and the revenue to be generated are equal. Such an analysis can be used to choose between different proposals at different estimated costs and revenue. Given estimated costs and revenue of two or more proposals, break-even analysis helps in choosing among them.

4.8. Business Case

[1*, c3]

The business case is the consolidated information summarizing and explaining a business proposal from different perspectives for a decision maker (cost, benefit, risk, and so on). It is often used to assess the potential value of a product, which can be used as a basis in the investment decision-making process. As opposed to a mere profit-loss calculation, the business case is a "case" of plans and analyses that is owned by the product

manager and used in support of achieving the business objectives.

4.9. Multiple Attribute Evaluation

[1*, c26]

The topics discussed so far are used to make decisions based on a single decision criterion: money. The alternative with the best present worth, the best ROI, and so forth is the one selected. Aside from technical feasibility, money is almost always the most important decision criterion, but it's not always the only one. Quite often there are other criteria, other "attributes," that need to be considered, and those attributes can't be cast in terms of money. Multiple attribute decision techniques allow other, nonfinancial criteria to be factored into the decision.

There are two families of multiple attribute decision techniques that differ in how they use the attributes in the decision. One family is the "compensatory," or single-dimensioned, techniques. This family collapses all of the attributes onto a single figure of merit. The family is called compensatory because, for any given alternative, a lower score in one attribute can be compensated by—or traded off against—a higher score in other attributes. The compensatory techniques include

- nondimensional scaling
- additive weighting
- analytic hierarchy process.

In contrast, the other family is the "noncompensatory," or fully dimensioned, techniques. This family does not allow tradeoffs among the attributes. Each attribute is treated as a separate entity in the decision process. The noncompensatory techniques include

- dominance
- satisficing
- lexicography.

4.10. Optimization Analysis

[1*, c20]

The typical use of optimization analysis is to study a cost function over a range of values to find the point where overall performance is best. Software's classic space-time tradeoff is an example of optimization; an algorithm that runs faster will often use more memory. Optimization balances the value of the faster runtime against the cost of the additional memory.

Real options analysis can be used to quantify the value of project choices, including the value of delaying a decision. Such options are difficult to compute with precision. However, awareness that choices have a monetary value provides insight in the timing of decisions such as increasing project staff or lengthening time to market to improve quality.

5. Practical Considerations

5.1. The "Good Enough" Principle

[1*, c21]

Often software engineering projects and products are not precise about the targets that should be achieved. Software requirements are stated, but the marginal value of adding a bit more functionality cannot be measured. The result could be late delivery or too-high cost. The "good enough" principle relates marginal value to marginal cost and provides guidance to determine criteria when a deliverable is "good enough" to be delivered. These criteria depend on business objectives and on prioritization of different alternatives, such as ranking software requirements, measurable quality attributes, or relating schedule to product content and cost.

The RACE principle (reduce accidents and control essence) is a popular rule towards good enough software. Accidents imply unnecessary overheads such as gold-plating and rework due to late defect removal or too many requirements changes. Essence is what customers pay for. Software engineering economics provides the mechanisms to define criteria that determine when a deliverable is "good enough" to be delivered. It also highlights that both words are relevant: "good" and "enough." Insufficient quality or insufficient quantity is not good enough.

Agile methods are examples of "good enough" that try to optimize value by reducing the overhead of delayed rework and the gold plating that

results from adding features that have low marginal value for the users (see Agile Methods in the Software Engineering Models and Methods KA and Software Life Cycle Models in the Software Engineering Process KA). In agile methods, detailed planning and lengthy development phases are replaced by incremental planning and frequent delivery of small increments of a deliverable product that is tested and evaluated by user representatives.

5.2. Friction-Free Economy

Economic friction is everything that keeps markets from having perfect competition. It involves distance, cost of delivery, restrictive regulations, and/or imperfect information. In high-friction markets, customers don't have many suppliers from which to choose. Having been in a business for a while or owning a store in a good location determines the economic position. It's hard for new competitors to start business and compete. The marketplace moves slowly and predictably. Friction-free markets are just the reverse. New competitors emerge and customers are quick to respond. The marketplace is anything but predictable. Theoretically, software and IT are friction-free. New companies can easily create products and often do so at a much lower cost than established companies, since they need not consider any legacies. Marketing and sales can be done via the Internet and social networks, and basically free distribution mechanisms can enable a ramp up to a global business. Software engineering economics aims to provide foundations to judge how a software business performs and how friction-free a market actually is. For instance, competition among software app developers is inhibited when apps must be sold through an app store and comply with that store's rules.

5.3. Ecosystems

An ecosystem is an environment consisting of all the mutually dependent stakeholders, business units, and companies working in a particular area. In a typical ecosystem, there are producers and consumers, where the consumers add value to the consumed resources. Note that a consumer is not the end user but an organization that uses the product to enhance it. A software ecosystem is, for instance, a supplier of an application working with companies doing the installation and support in different regions. Neither one could exist without the other. Ecosystems can be permanent or temporary. Software engineering economics provides the mechanisms to evaluate alternatives in establishing or extending an ecosystem—for instance, assessing whether to work with a specific distributor or have the distribution done by a company doing service in an area.

5.4. Offshoring and Outsourcing

Offshoring means executing a business activity beyond sales and marketing outside the home country of an enterprise. Enterprises typically either have their offshoring branches in low-cost countries or they ask specialized companies abroad to execute the respective activity. Offshoring should therefore not be confused with outsourcing. Offshoring within a company is called captive offshoring. Outsourcing is the result-oriented relationship with a supplier who executes business activities for an enterprise when, traditionally, those activities were executed inside the enterprise. Outsourcing is site-independent. The supplier can reside in the neighborhood of the enterprise or offshore (outsourced offshoring). Software engineering economics provides the basic criteria and business tools to evaluate different sourcing mechanisms and control their performance. For instance, using an outsourcing supplier for software development and maintenance might reduce the cost per hour of software development, but increase the number of hours and capital expenses due to an increased need for monitoring and communication. (For more information on offshoring and outsourcing, see "Outsourcing" in Management Issues in the Software Maintenance KA.)

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	Tockey 2005 [1*]	Sommerville 2011 [2*]	Fairley 2009 [3*]
1. Software Engineering Economics Fundamentals			
1.1. Finance	c2		
1.2. Accounting	c15		
1.3. Controlling	c15		
1.4. Cash Flow	c3		
1.5. Decision-Making Process	c2, c4		
1.6. Valuation	c5, c8		
1.7. Inflation	c13		
1.8. Depreciation	c14		
1.9. Taxation	c16, c17		
1.10. Time-Value of Money	c5, c11		
1.11. Efficiency		c1	
1.12. Effectiveness		c1	
1.13. Productivity		c23	
2. Life Cycle Economics			
2.1. Product		c22	c6
2.2. Project		c22	c1
2.3. Program			
2.4. Portfolio			
2.5. Product Life Cycle		c2	c2
2.6. Project Life Cycle		c2	c2
2.7. Proposals	c3		
2.8. Investment Decisions	c4		
2.9. Planning Horizon	c11		
2.10. Price and Pricing	c13		
2.11. Cost and Costing	c15		
2.12. Performance Measurement			c7, c8
2.13. Earned Value Management			c8
2.14. Termination Decisions	c11, c12	c9	
2.15. Replacement and Retirement Decisions	c12	c9	

	Tockey 2005 [1*]	Sommerville 2011 [2*]	Fairley 2009 [3*]
3. Risk and Uncertainty			
3.1. Goals, Estimates, and Plans			c6
3.2. Estimation Techniques			c6
3.3. Addressing Uncertainty			c6
3.4. Prioritization			c6
3.5. Decisions under Risk	c24		c9
3.6. Decisions under Uncertainty	c25		c9
4. Economic Analysis Methods			
4.1. For-Profit Decision Analysis	c10		
4.2. Minimum Acceptable Rate of Return	c10		
4.3. Return on Investment	c10		
4.4. Return on Capital Employed			
4.5. Cost-Benefit Analysis	c18		
4.6. Cost-Effectiveness Analysis	c18		
4.7. Break-Even Analysis	c19		
4.8. Business Case	c3		
4.9. Multiple Attribute Evaluation	c26		
4.10. Optimization Analysis	c20		
5. Practical Considerations			
5.1. The “Good Enough” Principle	c21		
5.2. Friction-Free Economy			
5.3. Ecosystems			
5.4. Offshoring and Outsourcing			

FURTHER READINGS

A Guide to the Project Management Body of Knowledge (PMBOK® Guide) [4].

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.

Software Extension to the Guide to the Project Management Body of Knowledge (SWX) [5].

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 description of processes that are applicable for managing adaptive life cycle software projects.

B.W. Boehm, *Software Engineering Economics* [6].

This book is the classic reading on software engineering economics. It provides an overview of business thinking in software engineering. Although the examples and figures are dated, it still is worth reading.

C. Ebert and R. Dumke, *Software Measurement* [7].

This book provides an overview on quantitative methods in software engineering, starting with measurement theory and proceeding to performance management and business decision making.

D.J. Reifer, *Making the Software Business Case: Improvement by the Numbers* [8].

This book is a classic reading on making a business case in the software and IT businesses. Many useful examples illustrate how the business case is formulated and quantified.

REFERENCES

[1*] S. Tockey, *Return on Software: Maximizing the Return on Your Software Investment*, Addison-Wesley, 2004.

[2*] J.H. Allen et al., *Software Security Engineering: A Guide for Project Managers*, Addison-Wesley, 2008.

[3*] R.E. Fairley, *Managing and Leading Software Projects*, Wiley-IEEE Computer Society Press, 2009.

[4] Project Management Institute, *A Guide to the Project Management Body of Knowledge (PMBOK(R) Guide)*, 5th ed., Project Management Institute, 2013.

[5] Project Management Institute and IEEE Computer Society, *Software Extension to the PMBOK® Guide Fifth Edition*, ed: Project Management Institute, 2013.

[6] B.W. Boehm, *Software Engineering Economics*, Prentice-Hall, 1981.

[7] C. Ebert and R. Dumke, *Software Measurement*, Springer, 2007.

[8] D.J. Reifer, *Making the Software Business Case: Improvement by the Numbers*, Addison Wesley, 2002.

CHAPTER 13

COMPUTING FOUNDATIONS

ACRONYMS

AOP	Aspect-Oriented Programming
ALU	Arithmetic and Logic Unit
API	Application Programming Interface
ATM	Asynchronous Transfer Mode
B/S	Browser-Server
CERT	Computer Emergency Response Team
COTS	Commercial Off-The-Shelf
CRUD	Create, Read, Update, Delete
C/S	Client-Server
CS	Computer Science
DBMS	Database Management System
FPU	Float Point Unit
I/O	Input and Output
ISA	Instruction Set Architecture
ISO	International Organization for Standardization
ISP	Internet Service Provider
LAN	Local Area Network
MUX	Multiplexer
NIC	Network Interface Card
OOP	Object-Oriented Programming
OS	Operating System
OSI	Open Systems Interconnection
PC	Personal Computer
PDA	Personal Digital Assistant
PPP	Point-to-Point Protocol
RFID	Radio Frequency Identification
RAM	Random Access Memory
ROM	Read Only Memory
SCSI	Small Computer System Interface
SQL	Structured Query Language
TCP	Transport Control Protocol
UDP	User Datagram Protocol
VPN	Virtual Private Network
WAN	Wide Area Network

INTRODUCTION

The scope of the Computing Foundations knowledge area (KA) encompasses the development and operational environment in which software evolves and executes. Because no software can exist in a vacuum or run without a computer, the core of such an environment is the computer and its various components. Knowledge about the computer and its underlying principles of hardware and software serves as a framework on which software engineering is anchored. Thus, all software engineers must have good understanding of the Computing Foundations KA.

It is generally accepted that software engineering builds on top of computer science. For example, "Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering" [1] clearly states, "One particularly important aspect is that *software engineering builds on computer science* and mathematics" (italics added).

Steve Tockey wrote in his book *Return on Software*:

> Both computer science and software engineering deal with computers, computing, and software. The science of computing, as a body of knowledge, is at the core of both.

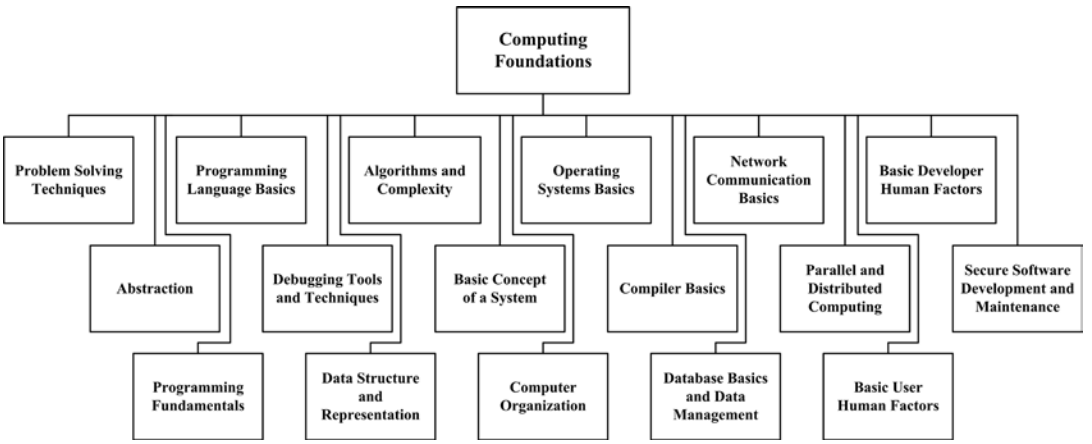

Figure 13.1. Breakdown of Topics for the Computing Foundations KA

> … Software engineering is concerned with the application of computers, computing, and software to practical purposes, specifically the design, construction, and operation of efficient and economical software systems.

Thus, at the core of software engineering is an understanding of computer science.

While few people will deny the role computer science plays in the development of software engineering both as a discipline and as a body of knowledge, the importance of computer science to software engineering cannot be overemphasized; thus, this Computing Foundations KA is being written.

The majority of topics discussed in the Computing Foundations KA are also topics of discussion in basic courses given in computer science undergraduate and graduate programs. Such courses include programming, data structure, algorithms, computer organization, operating systems, compilers, databases, networking, distributed systems, and so forth. Thus, when breaking down topics, it can be tempting to decompose the Computing Foundations KA according to these often-found divisions in relevant courses.

However, a purely course-based division of topics suffers serious drawbacks. For one, not all courses in computer science are related or equally important to software engineering. Thus, some topics that would otherwise be covered in a computer science course are not covered in this KA. For example, computer graphics—while an important course in a computer science degree program—is not included in this KA.

Second, some topics discussed in this guideline do not exist as standalone courses in undergraduate or graduate computer science programs. Consequently, such topics may not be adequately covered in a purely course-based breakdown. For example, abstraction is a topic incorporated into several different computer science courses; it is unclear which course abstraction should belong to in a course-based breakdown of topics.

The Computing Foundations KA is divided into seventeen different topics. A topic's direct usefulness to software engineers is the criterion used for selecting topics for inclusion in this KA (see Figure 13.1). The advantage of this topic-based breakdown is its foundation on the belief that Computing Foundations—if it is to be grasped firmly—must be considered as a collection of logically connected topics undergirding software engineering in general and software construction in particular.

The Computing Foundations KA is related closely to the Software Design, Software Construction, Software Testing, Software Maintenance, Software Quality, and Mathematical Foundations KAs.

BREAKDOWN OF TOPICS FOR COMPUTING FOUNDATIONS

The breakdown of topics for the Computing Foundations KA is shown in Figure 13.1.

1. Problem Solving Techniques

[2*, s3.2, c4] [3*, c5]

The concepts, notions, and terminology introduced here form an underlying basis for understanding the role and scope of problem solving techniques.

1.1. Definition of Problem Solving

Problem solving refers to the thinking and activities conducted to answer or derive a solution to a problem. There are many ways to approach a problem, and each way employs different tools and uses different processes. These different ways of approaching problems gradually expand and define themselves and finally give rise to different disciplines. For example, software engineering focuses on solving problems using computers and software.

While different problems warrant different solutions and may require different tools and processes, the methodology and techniques used in solving problems do follow some guidelines and can often be generalized as problem solving techniques. For example, a general guideline for solving a generic engineering problem is to use the three-step process given below [2*].

- Formulate the real problem.
- Analyze the problem.
- Design a solution search strategy.

1.2. Formulating the Real Problem

Gerard Voland writes, "It is important to recognize that a specific problem should be formulated if one is to develop a specific solution" [2*]. This formulation is called the problem statement, which explicitly specifies what both the problem and the desired outcome are.

Although there is no universal way of stating a problem, in general a problem should be expressed in such a way as to facilitate the development of solutions. Some general techniques to help one formulate the real problem include statement-restatement, determining the source and the cause, revising the statement, analyzing present and desired state, and using the fresh eye approach.

1.3. Analyze the Problem

Once the problem statement is available, the next step is to analyze the problem statement or situation to help structure our search for a solution. Four types of analysis include *situation analysis,* in which the most urgent or critical aspects of a situation are identified first; *problem analysis,* in which the cause of the problem must be determined; *decision analysis,* in which the action(s) needed to correct the problem or eliminate its cause must be determined; and *potential problem analysis,* in which the action(s) needed to prevent any reoccurrences of the problem or the development of new problems must be determined.

1.4. Design a Solution Search Strategy

Once the problem analysis is complete, we can focus on structuring a search strategy to find the solution. In order to find the "best" solution (here, "best" could mean different things to different people, such as faster, cheaper, more usable, different capabilities, etc.), we need to eliminate paths that do not lead to viable solutions, design tasks in a way that provides the most guidance in searching for a solution, and use various attributes of the final solution state to guide our choices in the problem solving process.

1.5. Problem Solving Using Programs

The uniqueness of computer software gives problem solving a flavor that is distinct from general engineering problem solving. To solve a problem using computers, we must answer the following questions.

- How do we figure out what to tell the computer to do?
- How do we convert the problem statement into an algorithm?
- How do we convert the algorithm into machine instructions?

The first task in solving a problem using a computer is to determine what to tell the computer to do. There may be many ways to tell the story, but all should take the perspective of a computer such

that the computer can eventually solve the problem. In general, a problem should be expressed in such a way as to facilitate the development of algorithms and data structures for solving it.

The result of the first task is a problem statement. The next step is to convert the problem statement into algorithms that solve the problem. Once an algorithm is found, the final step converts the algorithm into machine instructions that form the final solution: software that solves the problem.

Abstractly speaking, problem solving using a computer can be considered as a process of problem transformation—in other words, the step-by-step transformation of a problem statement into a problem solution. To the discipline of software engineering, the ultimate objective of problem solving is to transform a problem expressed in natural language into electrons running around a circuit. In general, this transformation can be broken into three phases:

a) Development of algorithms from the problem statement.
b) Application of algorithms to the problem.
c) Transformation of algorithms to program code.

The conversion of a problem statement into algorithms and algorithms into program codes usually follows a "stepwise refinement" (a.k.a. systematic decomposition) in which we start with a problem statement, rewrite it as a task, and recursively decompose the task into a few simpler subtasks until the task is so simple that solutions to it are straightforward. There are three basic ways of decomposing: sequential, conditional, and iterative.

2. Abstraction

[3*, s5.2–5.4]

Abstraction is an indispensible 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).

2.1. 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.

2.2. 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 of the 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.

2.3. 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.

2.4. 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 understanding the problem and its solution. Though beneficial, it is as times difficult to keep alternate abstractions in sync.

3. Programming Fundamentals

[3*, c6–19]

Programming is composed of the methodologies or activities for creating computer programs that perform a desired function. It is an indispensible part in software construction. In general, programming can be considered as the process of designing, writing, testing, debugging, and maintaining the source code. This source code is written in a programming language.

The process of writing source code often requires expertise in many different subject areas—including knowledge of the application domain, appropriate data structures, specialized algorithms, various language constructs, good programming techniques, and software engineering.

3.1. The Programming Process

Programming involves design, writing, testing, debugging, and maintenance. *Design* is the conception or invention of a scheme for turning a customer requirement for computer software into operational software. It is the activity that links application requirements to coding and debugging. *Writing* is the actual coding of the design in an appropriate programming language. *Testing* is the activity to verify that the code one writes actually does what it is supposed to do. *Debugging* is the activity to find and fix bugs (faults) in the source code (or design). *Maintenance* is the activity to update, correct, and enhance existing programs. Each of these activities is a huge topic and often warrants the explanation of an entire KA in the *SWEBOK Guide* and many books.

3.2. Programming Paradigms

Programming is highly creative and thus somewhat personal. Different people often write different programs for the same requirements. This diversity of programming causes much difficulty in the construction and maintenance of large complex software. Various programming paradigms have been developed over the years to put some standardization into this highly creative and personal activity. When one programs, he or she can use one of several programming paradigms to write the code. The major types of programming paradigms are discussed below.

Unstructured Programming: In unstructured programming, a programmer follows his/her

hunch to write the code in whatever way he/she likes as long as the function is operational. Often, the practice is to write code to fulfill a specific utility without regard to anything else. Programs written this way exhibit no particular structure—thus the name "unstructured programming." Unstructured programming is also sometimes called ad hoc programming.

Structured/Procedural/ Imperative Programming: A hallmark of structured programming is the use of well-defined control structures, including procedures (and/or functions) with each procedure (or function) performing a specific task. Interfaces exist between procedures to facilitate correct and smooth calling operations of the programs. Under structured programming, programmers often follow established protocols and rules of thumb when writing code. These protocols and rules can be numerous and cover almost the entire scope of programming—ranging from the simplest issue (such as how to name variables, functions, procedures, and so forth) to more complex issues (such as how to structure an interface, how to handle exceptions, and so forth).

Object-Oriented Programming: While procedural programming organizes programs around procedures, object-oriented programming (OOP) organize a program around objects, which are abstract data structures that combine both data and methods used to access or manipulate the data. The primary features of OOP are that objects representing various abstract and concrete entities are created and these objects interact with each other to collectively fulfill the desired functions.

Aspect-Oriented Programming: Aspect-oriented programming (AOP) is a programming paradigm that is built on top of OOP. AOP aims to isolate secondary or supporting functions from the main program's business logic by focusing on the cross sections (concerns) of the objects. The primary motivation for AOP is to resolve the object tangling and scattering associated with OOP, in which the interactions among objects become very complex. The essence of AOP is the greatly emphasized separation of concerns, which separates noncore functional concerns or logic into various aspects.

Functional Programming: Though less popular, functional programming is as viable as the other paradigms in solving programming problems. In functional programming, all computations are treated as the evaluation of mathematical functions. In contrast to the imperative programming that emphasizes changes in state, functional programming emphasizes the application of functions, avoids state and mutable data, and provides referential transparency.

4. Programming Language Basics

[4*, c6]

Using computers to solve problems involves programming—which is writing and organizing instructions telling the computer what to do at each step. Programs must be written in some programming language with which and through which we describe necessary computations. In other words, we use the facilities provided by a programming language to describe problems, develop algorithms, and reason about problem solutions. To write any program, one must understand at least one programming language.

4.1. Programming Language Overview

A programming language is designed to express computations that can be performed by a computer. In a practical sense, a programming language is a notation for writing programs and thus should be able to express most data structures and algorithms. Some, but not all, people restrict the term "programming language" to those languages that can express all possible algorithms.

Not all languages have the same importance and popularity. The most popular ones are often defined by a specification document established by a well-known and respected organization. For example, the C programming language is specified by an ISO standard named ISO/IEC 9899. Other languages, such as Perl and Python, do not enjoy such treatment and often have a dominant implementation that is used as a reference.

4.2. Syntax and Semantics of Programming Languages

Just like natural languages, many programming languages have some form of written specification of their syntax (form) and semantics (meaning). Such specifications include, for example,

specific requirements for the definition of variables and constants (in other words, declaration and types) and format requirements for the instructions themselves.

In general, a programming language supports such constructs as variables, data types, constants, literals, assignment statements, control statements, procedures, functions, and comments. The syntax and semantics of each construct must be clearly specified.

4.3. Low-Level Programming Languages

Programming language can be classified into two classes: low-level languages and high-level languages. Low-level languages can be understood by a computer with no or minimal assistance and typically include machine languages and assembly languages. A machine language uses ones and zeros to represent instructions and variables, and is directly understandable by a computer. An assembly language contains the same instructions as a machine language but the instructions and variables have symbolic names that are easier for humans to remember.

Assembly languages cannot be directly understood by a computer and must be translated into a machine language by a utility program called an *assembler.* There often exists a correspondence between the instructions of an assembly language and a machine language, and the translation from assembly code to machine code is straightforward. For example, "add r1, r2, r3" is an assembly instruction for adding the content of register r2 and r3 and storing the sum into register r1. This instruction can be easily translated into machine code "0001 0001 0010 0011**.**" (Assume the operation code for addition is 0001, see Figure 13.2).

add	r1,	r2,	r3
0001	0001	0010	0011

Figure 13.2. Assembly-to-Binary Translations

One common trait shared by these two types of language is their close association with the specifics of a type of computer or instruction set architecture (ISA).

4.4. High-Level Programming Languages

A high-level programming language has a strong abstraction from the details of the computer's ISA. In comparison to low-level programming languages, it often uses natural-language elements and is thus much easier for humans to understand. Such languages allow symbolic naming of variables, provide expressiveness, and enable abstraction of the underlying hardware. For example, while each microprocessor has its own ISA, code written in a high-level programming language is usually portable between many different hardware platforms. For these reasons, most programmers use and most software are written in high-level programming languages. Examples of high-level programming languages include C, C++, C#, and Java.

4.5. Declarative vs. Imperative Programming Languages

Most programming languages (high-level or low-level) allow programmers to specify the individual instructions that a computer is to execute. Such programming languages are called imperative programming languages because one has to specify every step clearly to the computer. But some programming languages allow programmers to only describe the function to be performed without specifying the exact instruction sequences to be executed. Such programming languages are called declarative programming languages. Declarative languages are high-level languages. The actual implementation of the computation written in such a language is hidden from the programmers and thus is not a concern for them.

The key point to note is that declarative programming only describes *what* the program should accomplish without describing *how* to accomplish it. For this reason, many people believe declarative programming facilitates easier software development. Declarative programming languages include Lisp (also a functional programming language) and Prolog, while imperative programming languages include C, C++, and JAVA.

5. Debugging Tools and Techniques

[3*, c23]

Once a program is coded and compiled (compilation will be discussed in section 10), the next step is debugging, which is a methodical process of finding and reducing the number of bugs or faults in a program. The purpose of debugging is to find out why a program doesn't work or produces a wrong result or output. Except for very simple programs, debugging is always necessary.

5.1. Types of Errors

When a program does not work, it is often because the program contains bugs or errors that can be either syntactic errors, logical errors, or data errors. Logical errors and data errors are also known as two categories of "faults" in software engineering terminology (see topic 1.1, Testing-Related Terminology, in the Software Testing KA).

Syntax errors are simply any error that prevents the translator (compiler/interpreter) from successfully parsing the statement. Every statement in a program must be parse-able before its meaning can be understood and interpreted (and, therefore, executed). In high-level programming languages, syntax errors are caught during the compilation or translation from the high-level language into machine code. For example, in the C/C++ programming language, the statement "123=constant;" contains a syntax error that will be caught by the compiler during compilation.

Logic errors are semantic errors that result in incorrect computations or program behaviors. Your program is legal, but wrong! So the results do not match the problem statement or user expectations. For example, in the C/C++ programming language, the inline function "int f(int *x*) {return f(*x*-1);}" for computing factorial x! is legal but logically incorrect. This type of error cannot be caught by a compiler during compilation and is often discovered through tracing the execution of the program (Modern static checkers do identify some of these errors. However, the point remains that these are not machine checkable in general).

Data errors are input errors that result either in input data that is different from what the program expects or in the processing of wrong data.

5.2. Debugging Techniques

Debugging involves many activities and can be static, dynamic, or postmortem. *Static debugging* usually takes the form of code review, while *dynamic debugging* usually takes the form of tracing and is closely associated with testing. *Postmortem debugging* is the act of debugging the core dump (memory dump) of a process. Core dumps are often generated after a process has terminated due to an unhandled exception. All three techniques are used at various stages of program development.

The main activity of dynamic debugging is tracing, which is executing the program one piece at a time, examining the contents of registers and memory, in order to examine the results at each step. There are three ways to trace a program.

- *Single-stepping:* execute one instruction at a time to make sure each instruction is executed correctly. This method is tedious but useful in verifying each step of a program.
- *Breakpoints:* tell the program to stop executing when it reaches a specific instruction. This technique lets one quickly execute selected code sequences to get a high-level overview of the execution behavior.
- *Watch points:* tell the program to stop when a register or memory location changes or when it equals to a specific value. This technique is useful when one doesn't know where or when a value is changed and when this value change likely causes the error.

5.3. Debugging Tools

Debugging can be complex, difficult, and tedious. Like programming, debugging is also highly creative (sometimes more creative than programming). Thus some help from tools is in order. For dynamic debugging, *debuggers* are widely used and enable the programmer to monitor the execution of a program, stop the execution, restart the execution, set breakpoints, change values in memory, and even, in some cases, go back in time.

For static debugging, there are many *static code analysis tools*, which look for a specific set of known problems within the source code.

Both commercial and free tools exist in various languages. These tools can be extremely useful when checking very large source trees, where it is impractical to do code walkthroughs. The UNIX *lint* program is an early example.

6. Data Structure and Representation

[5*, s2.1–2.6]

Programs work on data. But data must be expressed and organized within computers before being processed by programs. This organization and expression of data for programs' use is the subject of data structure and representation. Simply put, a data structure tries to store and organize data in a computer in such a way that the data can be used efficiently. There are many types of data structures and each type of structure is suitable for some kinds of applications. For example, B/B+ trees are well suited for implementing massive file systems and databases.

6.1. Data Structure Overview

Data structures are computer representations of data. Data structures are used in almost every program. In a sense, no meaningful program can be constructed without the use of some sort of data structure. Some design methods and programming languages even organize an entire software system around data structures. Fundamentally, data structures are abstractions defined on a collection of data and its associated operations.

Often, data structures are designed for improving program or algorithm efficiency. Examples of such data structures include stacks, queues, and heaps. At other times, data structures are used for conceptual unity (abstract data type), such as the name and address of a person. Often, a data structure can determine whether a program runs in a few seconds or in a few hours or even a few days.

From the perspective of physical and logical ordering, a data structure is either linear or nonlinear. Other perspectives give rise to different classifications that include homogeneous vs. heterogeneous, static vs. dynamic, persistent vs. transient, external vs. internal, primitive vs. aggregate, recursive vs. nonrecursive; passive vs. active; and stateful vs. stateless structures.

6.2. Types of Data Structure

As mentioned above, different perspectives can be used to classify data structures. However, the predominant perspective used in classification centers on physical and logical ordering between data items. This classification divides data structures into linear and nonlinear structures. Linear structures organize data items in a single dimension in which each data entry has one (physical or logical) predecessor and one successor with the exception of the first and last entry. The first entry has no predecessor and the last entry has no successor. Nonlinear structures organize data items in two or more dimensions, in which case one entry can have multiple predecessors and successors. Examples of linear structures include lists, stacks, and queues. Examples of nonlinear structures include heaps, hash tables, and trees (such as binary trees, balance trees, B-trees, and so forth).

Another type of data structure that is often encountered in programming is the compound structure. A compound data structure builds on top of other (more primitive) data structures and, in some way, can be viewed as the same structure as the underlying structure. Examples of compound structures include sets, graphs, and partitions. For example, a partition can be viewed as a set of sets.

6.3. Operations on Data Structures

All data structures support some operations that produce a specific structure and ordering, or retrieve relevant data from the structure, store data into the structure, or delete data from the structure. Basic operations supported by all data structures include create, read, update, and delete (CRUD).

- Create: Insert a new data entry into the structure.
- Read: Retrieve a data entry from the structure.
- Update: Modify an existing data entry.
- Delete: Remove a data entry from the structure.

Some data structures also support additional operations:

- Find a particular element in the structure.
- Sort all elements according to some ordering.
- Traverse all elements in some specific order.
- Reorganize or rebalance the structure.

Different structures support different operations with different efficiencies. The difference between operation efficiency can be significant. For example, it is easy to retrieve the last item inserted into a stack, but finding a particular element within a stack is rather slow and tedious.

7. Algorithms and Complexity

[5*, s1.1–1.3, s3.3–3.6, s4.1–4.8, s5.1–5.7, s6.1–6.3, s7.1–7.6, s11.1, s12.1]

Programs are not random pieces of code: they are meticulously written to perform user-expected actions. The guide one uses to compose programs are algorithms, which organize various functions into a series of steps and take into consideration the application domain, the solution strategy, and the data structures being used. An algorithm can be very simple or very complex.

7.1. Overview of Algorithms

Abstractly speaking, algorithms guide the operations of computers and consist of a sequence of actions composed to solve a problem. Alternative definitions include but are not limited to:

- An algorithm is any well-defined computational procedure that takes some value or set of values as input and produces some value or set of values as output.
- An algorithm is a sequence of computational steps that transform the input into the output.
- An algorithm is a tool for solving a well-specified computation problem.

Of course, different definitions are favored by different people. Though there is no universally accepted definition, some agreement exists that an algorithm needs to be correct, finite (in other words, terminate eventually or one must be able to write it in a finite number of steps), and unambiguous.

7.2. Attributes of Algorithms

The attributes of algorithms are many and often include modularity, correctness, maintainability, functionality, robustness, user-friendliness (i.e. easy to be understood by people), programmer time, simplicity, and extensibility. A commonly emphasized attribute is "performance" or "efficiency" by which we mean both time and resource-usage efficiency while generally emphasizing the time axis. To some degree, efficiency determines if an algorithm is feasible or impractical. For example, an algorithm that takes one hundred years to terminate is virtually useless and is even considered incorrect.

7.3. Algorithmic Analysis

Analysis of algorithms is the theoretical study of computer-program performance and resource usage; to some extent it determines the goodness of an algorithm. Such analysis usually abstracts away the particular details of a specific computer and focuses on the asymptotic, machine-independent analysis.

There are three basic types of analysis. In *worst-case analysis,* one determines the maximum time or resources required by the algorithm on any input of size n. In *average-case analysis,* one determines the expected time or resources required by the algorithm over all inputs of size n; in performing average-case analysis, one often needs to make assumptions on the statistical distribution of inputs. The third type of analysis is the *best-case analysis,* in which one determines the minimum time or resources required by the algorithm on any input of size n. Among the three types of analysis, average-case analysis is the most relevant but also the most difficult to perform.

Besides the basic analysis methods, there are also the *amortized analysis,* in which one determines the maximum time required by an algorithm over a sequence of operations; and the *competitive analysis,* in which one determines the relative performance merit of an algorithm against the optimal algorithm (which may not be known) in the same category (for the same operations).

7.4. Algorithmic Design Strategies

The design of algorithms generally follows one of the following strategies: brute force, divide and conquer, dynamic programming, and greedy selection. The *brute force strategy* is actually a no-strategy. It exhaustively tries every possible way to tackle a problem. If a problem has a solution, this strategy is guaranteed to find it; however, the time expense may be too high. The *divide and conquer strategy* improves on the brute force strategy by dividing a big problem into smaller, homogeneous problems. It solves the big problem by recursively solving the smaller problems and combing the solutions to the smaller problems to form the solution to the big problem. The underlying assumption for divide and conquer is that smaller problems are easier to solve.

The *dynamic programming strategy* improves on the divide and conquer strategy by recognizing that some of the sub-problems produced by division may be the same and thus avoids solving the same problems again and again. This elimination of redundant subproblems can dramatically improve efficiency.

The *greedy selection strategy* further improves on dynamic programming by recognizing that not all of the sub-problems contribute to the solution of the big problem. By eliminating all but one sub-problem, the greedy selection strategy achieves the highest efficiency among all algorithm design strategies. Sometimes the use of *randomization* can improve on the greedy selection strategy by eliminating the complexity in determining the greedy choice through coin flipping or randomization.

7.5. Algorithmic Analysis Strategies

The analysis strategies of algorithms include *basic counting analysis,* in which one actually counts the number of steps an algorithm takes to complete its task; *asymptotic analysis,* in which one only considers the order of magnitude of the number of steps an algorithm takes to complete its task; *probabilistic analysis,* in which one makes use of probabilities in analyzing the average performance of an algorithm; *amortized analysis,* in which one uses the methods of aggregation, potential, and accounting to analyze the worst performance of an algorithm on a sequence of operations; and *competitive analysis,* in which one uses methods such as potential and accounting to analyze the relative performance of an algorithm to the optimal algorithm.

For complex problems and algorithms, one may need to use a combination of the aforementioned analysis strategies.

8. Basic Concept of a System

[6*, c10]

Ian Sommerville writes, "a system is a purposeful collection of interrelated components that work together to achieve some objective" [6*]. A system can be very simple and include only a few components, like an ink pen, or rather complex, like an aircraft. Depending on whether humans are part of the system, systems can be divided into technical computer-based systems and sociotechnical systems. A technical computer-based system functions without human involvement, such as televisions, mobile phones, thermostat, and some software; a sociotechnical system will not function without human involvement. Examples of such system include manned space vehicles, chips embedded inside a human, and so forth.

8.1. Emergent System Properties

A system is more than simply the sum of its parts. Thus, the properties of a system are not simply the sum of the properties of its components. Instead, a system often exhibits properties that are properties of the system as a whole. These properties are called *emergent properties* because they develop only after the integration of constituent parts in the system. Emergent system properties can be either functional or nonfunctional. Functional properties describe the things that a system does. For example, an aircraft's functional properties include flotation on air, carrying people or cargo, and use as a weapon of mass destruction. Nonfunctional properties describe how the system behaves in its operational environment. These can include such qualities as consistency, capacity, weight, security, etc.

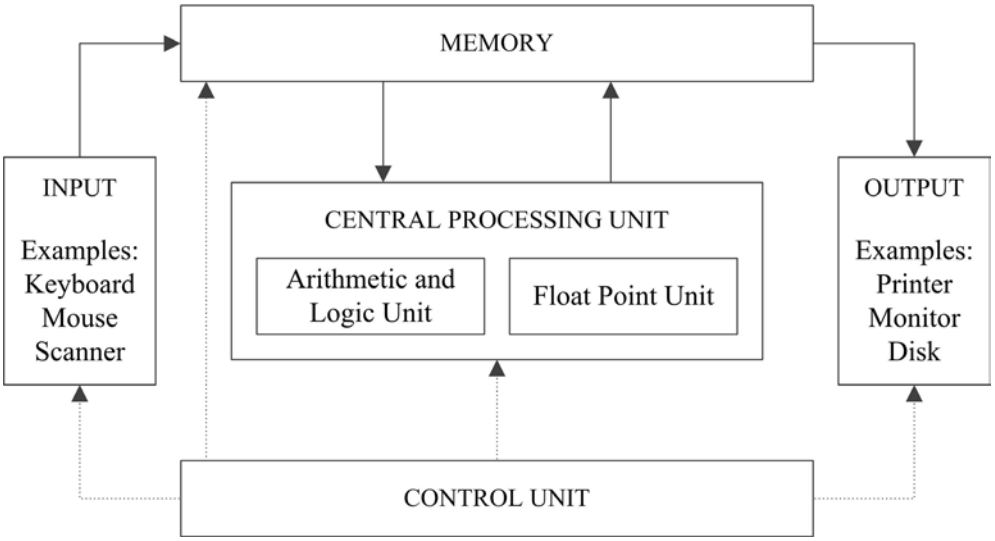

Figure 13.3. Basic Components of a Computer System Based on the von Neumann Model

8.2. Systems Engineering

"Systems engineering is the interdisciplinary approach governing the total technical and managerial effort required to transform a set of customer needs, expectations, and constraints into a solution and to support that solution throughout its life." [7]. The life cycle stages of systems engineering vary depending on the system being built but, in general, include system requirements definition, system design, sub-system development, system integration, system testing, system installation, system evolution, and system decommissioning.

Many practical guidelines have been produced in the past to aid people in performing the activities of each phase. For example, system design can be broken into smaller tasks of identification of subsystems, assignment of system requirements to subsystems, specification of subsystem functionality, definition of sub-system interfaces, and so forth.

8.3. Overview of a Computer System

Among all the systems, one that is obviously relevant to the software engineering community is the computer system. A computer is a machine that executes programs or software. It consists of a purposeful collection of mechanical, electrical, and electronic components with each component performing a preset function. Jointly, these components are able to execute the instructions that are given by the program.

Abstractly speaking, a computer receives some input, stores and manipulates some data, and provides some output. The most distinct feature of a computer is its ability to store and execute sequences of instructions called *programs.* An interesting phenomenon concerning the computer is the universal equivalence in functionality. According to Turing, all computers with a certain minimum capability are equivalent in their ability to perform computation tasks. In other words, given enough time and memory, all computers—ranging from a netbook to a supercomputer—are capable of computing exactly the same things, irrespective of speed, size, cost, or anything else.

Most computer systems have a structure that is known as the "von Neumann model," which consists of five components: a *memory* for storing instructions and data, a *central processing unit* for performing arithmetic and logical operations, a *control unit* for sequencing and interpreting instructions, *input* for getting external information into the memory, and *output* for producing results for the user. The basic components of a computer system based on the von Neumann model are depicted in Figure 13.3.

9. Computer Organization

[8*, c1–c4]

From the perspective of a computer, a wide semantic gap exists between its intended behavior and the workings of the underlying electronic devices that actually do the work within the computer. This gap is bridged through computer organization, which meshes various electrical, electronic, and mechanical devices into one device that forms a computer. The objects that computer organization deals with are the devices, connections, and controls. The abstraction built in computer organization is the computer.

9.1. Computer Organization Overview

A computer generally consists of a CPU, memory, input devices, and output devices. Abstractly speaking, the organization of a computer can be divided into four levels (Figure 13.4). The *macro architecture* level is the formal specification of all the functions a particular machine can carry out and is known as the instruction set architecture (ISA). The *micro architecture* level is the implementation of the ISA in a specific CPU—in other words, the way in which the ISA's specifications are actually carried out. The *logic circuits* level is the level where each functional component of the micro architecture is built up of circuits that make decisions based on simple rules. The *devices* level is the level where, finally, each logic circuit is actually built of electronic devices such as complementary metal-oxide semiconductors (CMOS), n-channel metal oxide semiconductors (NMOS), or gallium arsenide (GaAs) transistors, and so forth.

Macro Architecture Level (ISA)
Micro Architecture Level
Logic Circuits Level
Devices Level

Figure 13.4. Machine Architecture Levels

Each level provides an abstraction to the level above and is dependent on the level below. To a programmer, the most important abstraction is the ISA, which specifies such things as the native data types, instructions, registers, addressing modes, the memory architecture, interrupt and exception handling, and the I/Os. Overall, the ISA specifies the ability of a computer and what can be done on the computer with programming.

9.2. Digital Systems

At the lowest level, computations are carried out by the electrical and electronic devices within a computer. The computer uses circuits and memory to hold charges that represents the presence or absence of voltage. The presence of voltage is equal to a 1 while the absence of voltage is a zero. On disk the polarity of the voltage is represented by 0s and 1s that in turn represents the data stored. Everything—including instruction and data—is expressed or encoded using digital zeros and ones. In this sense, a computer becomes a digital system. For example, decimal value 6 can be encoded as 110, the addition instruction may be encoded as 0001, and so forth. The component of the computer such as the control unit, ALU, memory and I/O use the information to compute the instructions.

9.3. Digital Logic

Obviously, logics are needed to manipulate data and to control the operation of computers. This logic, which is behind a computer's proper function, is called *digital logic* because it deals with the operations of digital zeros and ones. Digital logic specifies the rules both for building various digital devices from the simplest elements (such as transistors) and for governing the operation of digital devices. For example, digital logic spells out what the value will be if a zero and one is ANDed, ORed, or exclusively ORed together. It also specifies how to build decoders, multiplexers (MUX), memory, and adders that are used to assemble the computer.

9.4. Computer Expression of Data

As mentioned before, a computer expresses data with electrical signals or digital zeros and ones. Since there are only two different digits used in

data expression, such a system is called a *binary expression system*. Due to the inherent nature of a binary system, the maximum numerical value expressible by an n-bits binary code is $2^n - 1$. Specifically, binary number $a_n a_{n-1} \ldots a_1 a_0$ corresponds to $a_n \times 2^n + a_{n-1} \times 2^{n-1} + \ldots + a_1 \times 2^1 + a_0 \times 2^0$. Thus, the numerical value of the binary expression of 1011 is $1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 = 11$. To express a nonnumerical value, we need to decide the number of zeros and ones to use and the order in which those zeros and ones are arranged.

Of course, there are different ways to do the encoding, and this gives rise to different data expression schemes and subschemes. For example, integers can be expressed in the form of unsigned, one's complement, or two's complement. For characters, there are ASCII, Unicode, and IBM's EBCDIC standards. For floating point numbers, there are IEEE-754 FP 1, 2, and 3 standards.

9.5. The Central Processing Unit (CPU)

The central processing unit is the place where instructions (or programs) are actually executed. The execution usually takes several steps, including fetching the program instruction, decoding the instruction, fetching operands, performing arithmetic and logical operations on the operands, and storing the result. The main components of a CPU consist of registers where instructions and data are often read from and written to, the arithmetic and logic unit (ALU) that performs the actual arithmetic (such as addition, subtraction, multiplication, and division) and logic (such as AND, OR, shift, and so forth) operations, the control unit that is responsible for producing proper signals to control the operations, and various (data, address, and control) buses that link the components together and transport data to and from these components.

9.6. Memory System Organization

Memory is the storage unit of a computer. It concerns the assembling of a large-scale memory system from smaller and single-digit storage units. The main topics covered by memory system architecture include the following:

- Memory cells and chips
- Memory boards and modules
- Memory hierarchy and cache
- Memory as a subsystem of the computer.

Memory cells and chips deal with single-digital storage and the assembling of single-digit units into one-dimensional memory arrays as well as the assembling of one-dimensional storage arrays into multi-dimensional storage memory chips. *Memory boards and modules* concern the assembling of memory chips into memory systems, with the focus being on the organization, operation, and management of the individual chips in the system. *Memory hierarchy and cache* are used to support efficient memory operations. *Memory as a sub-system* deals with the interface between the memory system and other parts of the computer.

9.7. Input and Output (I/O)

A computer is useless without I/O. Common input devices include the keyboard and mouse; common output devices include the disk, the screen, the printer, and speakers. Different I/O devices operate at different data rates and reliabilities. How computers connect and manage various input and output devices to facilitate the interaction between computers and humans (or other computers) is the focus of topics in I/O. The main issues that must be resolved in input and output are the ways I/O can and should be performed.

In general, I/O is performed at both hardware and software levels. Hardware I/O can be performed in any of three ways. *Dedicated I/O* dedicates the CPU to the actual input and output operations during I/O; *memory-mapped I/O* treats I/O operations as memory operations; and *hybrid I/O* combines dedicated I/O and memory-mapped I/O into a single holistic I/O operation mode.

Coincidentally, software I/O can also be performed in one of three ways. *Programmed I/O* lets the CPU wait while the I/O device is doing I/O; *interrupt-driven I/O* lets the CPU's handling of I/O be driven by the I/O device; and *direct memory access* (*DMA)* lets I/O be handled by a secondary CPU embedded in a DMA device (or

channel). (Except during the initial setup, the main CPU is not disturbed during a DMA I/O operation.)

Regardless of the types of I/O scheme being used, the main issues involved in I/O include *I/O addressing* (which deals with the issue of how to identify the I/O device for a specific I/O operation), *synchronization* (which deals with the issue of how to make the CPU and I/O device work in harmony during I/O), and *error detection and correction* (which deals with the occurrence of transmission errors).

10. Compiler Basics

[4*, s6.4] [8*, s8.4]

10.1. Compiler/Interpreter Overview

Programmers usually write programs in high level language code, which the CPU cannot execute; so this source code has to be converted into machine code to be understood by a computer. Due to the differences between different ISAs, the translation must be done for each ISA or specific machine language under consideration.

The translation is usually performed by a piece of software called a compiler or an interpreter. This process of translation from a high-level language to a machine language is called compilation, or, sometimes, interpretation.

10.2. Interpretation and Compilation

There are two ways to translate a program written in a higher-level language into machine code: interpretation and compilation. *Interpretation* translates the source code one statement at a time into machine language, executes it on the spot, and then goes back for another statement. Both the high-level-language source code and the interpreter are required every time the program is run.

Compilation translates the high-level-language source code into an entire machine-language program (an executable image) by a program called a compiler. After compilation, only the executable image is needed to run the program. Most application software is sold in this form.

While both compilation and interpretation convert high level language code into machine code, there are some important differences between the two methods. First, a compiler makes the conversion just once, while an interpreter typically converts it every time a program is executed. Second, interpreting code is slower than running the compiled code, because the interpreter must analyze each statement in the program when it is executed and then perform the desired action, whereas the compiled code just performs the action within a fixed context determined by the compilation. Third, access to variables is also slower in an interpreter because the mapping of identifiers to storage locations must be done repeatedly at runtime rather than at compile time.

The primary tasks of a compiler may include preprocessing, lexical analysis, parsing, semantic analysis, code generation, and code optimization. Program faults caused by incorrect compiler behavior can be very difficult to track down. For this reason, compiler implementers invest a lot of time ensuring the correctness of their software.

10.3. The Compilation Process

Compilation is a complex task. Most compilers divide the compilation process into many phases. A typical breakdown is as follows:

- Lexical Analysis
- Syntax Analysis or Parsing
- Semantic Analysis
- Code Generation

Lexical analysis partitions the input text (the source code), which is a sequence of characters, into separate *comments*, which are to be ignored in subsequent actions, and *basic symbols, which have lexical meanings.* These basic symbols must correspond to some terminal symbols of the grammar of the particular programming language. Here terminal symbols refer to the elementary symbols (or tokens) in the grammar that cannot be changed.

Syntax analysis is based on the results of the lexical analysis and discovers the structure in the program and determines whether or not a text conforms to an expected format. *Is this a textually correct C++ program?* or *Is this entry textually correct?* are typical questions that can be

answered by syntax analysis. Syntax analysis determines if the source code of a program is correct and converts it into a more structured representation (parse tree) for semantic analysis or transformation.

Semantic analysis adds semantic information to the parse tree built during the syntax analysis and builds the symbol table. It performs various semantic checks that include type checking, object binding (associating variable and function references with their definitions), and definite assignment (requiring all local variables to be initialized before use). If mistakes are found, the semantically incorrect program statements are rejected and flagged as errors.

Once semantic analysis is complete, the phase of *code generation* begins and transforms the intermediate code produced in the previous phases into the native machine language of the computer under consideration. This involves resource and storage decisions—such as deciding which variables to fit into registers and memory and the selection and scheduling of appropriate machine instructions, along with their associated addressing modes.

It is often possible to combine multiple phases into one pass over the code in a compiler implementation. Some compilers also have a preprocessing phase at the beginning or after the lexical analysis that does necessary housekeeping work, such as processing the program instructions for the compiler (directives). Some compilers provide an optional optimization phase at the end of the entire compilation to optimize the code (such as the rearrangement of instruction sequence) for efficiency and other desirable objectives requested by the users.

11. Operating Systems Basics

[4*, c3]

Every system of meaningful complexity needs to be managed. A computer, as a rather complex electrical-mechanical system, needs its own manager for managing the resources and activities occurring on it. That manager is called an *operating system* (OS).

11.1. Operating Systems Overview

Operating systems is a collection of software and firmware, that controls the execution of computer programs and provides such services as computer resource allocation, job control, input/output control, and file management in a computer system. Conceptually, an operating system is a computer program that manages the hardware resources and makes it easier to use by applications by presenting nice abstractions. This nice abstraction is often called the virtual machine and includes such things as processes, virtual memory, and file systems. An OS hides the complexity of the underlying hardware and is found on all modern computers.

The principal roles played by OSs are management and illusion. *Management* refers to the OS's management (allocation and recovery) of physical resources among multiple competing users/applications/tasks. *Illusion* refers to the nice abstractions the OS provides.

11.2. Tasks of an Operating System

The tasks of an operating system differ significantly depending on the machine and time of its invention. However, modern operating systems have come to agreement as to the tasks that must be performed by an OS. These tasks include CPU management, memory management, disk management (file system), I/O device management, and security and protection. Each OS task manages one type of physical resource.

Specifically, CPU management deals with the allocation and releases of the CPU among competing programs (called processes/threads in OS jargon), including the operating system itself. The main abstraction provided by CPU management is the process/thread model. Memory management deals with the allocation and release of memory space among competing processes, and the main abstraction provided by memory management is virtual memory. Disk management deals with the sharing of magnetic or optical or solid state disks among multiple programs/users and its main abstraction is the file system. I/O device management deals with the allocation and releases of various I/O devices among competing processes.

Security and protection deal with the protection of computer resources from illegal use.

11.3. Operating System Abstractions

The arsenal of OSs is abstraction. Corresponding to the five physical tasks, OSs use five abstractions: process/thread, virtual memory, file systems, input/output, and protection domains. The overall OS abstraction is the virtual machine.

For each task area of OS, there is both a physical reality and a conceptual abstraction. The physical reality refers to the hardware resource under management; the conceptual abstraction refers to the interface the OS presents to the users/programs above. For example, in the thread model of the OS, the physical reality is the CPU and the abstraction is multiple CPUs. Thus, a user doesn't have to worry about sharing the CPU with others when working on the abstraction provided by an OS. In the virtual memory abstraction of an OS, the physical reality is the physical RAM or ROM (whatever), the abstraction is multiple unlimited memory space. Thus, a user doesn't have to worry about sharing physical memory with others or about limited physical memory size.

Abstractions may be virtual or transparent; in this context virtual applies to something that appears to be there, but isn't (like usable memory beyond physical), whereas transparent applies to something that is there, but appears not to be there (like fetching memory contents from disk or physical memory).

11.4. Operating Systems Classification

Different operating systems can have different functionality implementation. In the early days of the computer era, operating systems were relatively simple. As time goes on, the complexity and sophistication of operating systems increases significantly. From a historical perspective, an operating system can be classified as one of the following.

- *Batching OS:* organizes and processes work in batches. Examples of such OSs include IBM's FMS, IBSYS, and University of Michigan's UMES.
- *Multiprogrammed batching OS:* adds multitask capability into earlier simple batching OSs. An example of such an OS is IBM's OS/360.
- *Time-sharing OS:* adds multi-task and interactive capabilities into the OS. Examples of such OSs include UNIX, Linux, and NT.
- *Real-time OS:* adds timing predictability into the OS by scheduling individual tasks according to each task's completion deadlines. Examples of such OS include VxWorks (WindRiver) and DART (EMC).
- *Distributed OS:* adds the capability of managing a network of computers into the OS.
- *Embedded OS:* has limited functionality and is used for embedded systems such as cars and PDAs. Examples of such OSs include Palm OS, Windows CE, and TOPPER.

Alternatively, an OS can be classified by its applicable target machine/environment into the following.

- *Mainframe OS:* runs on the mainframe computers and include OS/360, OS/390, AS/400, MVS, and VM.
- *Server OS:* runs on workstations or servers and includes such systems as UNIX, Windows, Linux, and VMS.
- *Multicomputer OS:* runs on multiple computers and include such examples as Novell Netware.
- *Personal computers OS:* runs on personal computers and include such examples as DOS, Windows, Mac OS, and Linux.
- *Mobile device OS:* runs on personal devices such as cell phones, IPAD and include such examples of iOS, Android, Symbian, etc.

12. Database Basics and Data Management

[4*, c9]

A database consists of an organized collection of data for one or more uses. In a sense, a database is a generalization and expansion of data structures. But the difference is that a database is usually external to individual programs and permanent in existence compared to data structures. Databases are used when the data volume is large or logical

relations between data items are important. The factors considered in database design include performance, concurrency, integrity, and recovery from hardware failures.

12.1. Entity and Schema

The things a database tries to model and store are called entities. Entities can be real-world objects such as persons, cars, houses, and so forth, or they may be abstract concepts such as persons, salary, names, and so forth. An entity can be primitive such as a name or composite such as an employee that consists of a name, identification number, salary, address, and so forth.

The single most important concept in a database is the *schema*, which is a description of the entire database structure from which all other database activities are built. A schema defines the relationships between the various entities that compose a database. For example, a schema for a company payroll system would consist of such things as employee ID, name, salary rate, address, and so forth. Database software maintains the database according to the schema.

Another important concept in database is the *database model* that describes the type of relationship among various entities. The commonly used models include relational, network, and object models.

12.2. Database Management Systems (DBMS)

Database Management System (DBMS) components include database applications for the storage of structured and unstructured data and the required database management functions needed to view, collect, store, and retrieve data from the databases. A DBMS controls the creation, maintenance, and use of the database and is usually categorized according to the database model it supports—such as the relational, network, or object model. For example, a relational database management system (RDBMS) implements features of the relational model. An object database management system (ODBMS) implements features of the object model.

12.3. Database Query Language

Users/applications interact with a database through a database query language, which is a specialized programming language tailored to database use. The database model tends to determine the query languages that are available to access the database. One commonly used query language for the relational database is the structured query language, more commonly abbreviated as SQL. A common query language for object databases is the object query language (abbreviated as OQL). There are three components of SQL: Data Definition Language (DDL), Data Manipulation Language (DML), and Data Control Language (DCL). An example of an DML query may look like the following:

```
SELECT Component_No, Quantity
FROM COMPONENT
WHERE Item_No = 100
```

The above query selects all the Component_No and its corresponding quantity from a database table called COMPONENT, where the Item_No equals to 100.

12.4. Tasks of DBMS Packages

A DBMS system provides the following capabilities:

- *Database development* is used to define and organize the content, relationships, and structure of the data needed to build a database.
- *Database interrogation* is used for accessing the data in a database for information retrieval and report generation. End users can selectively retrieve and display information and produce printed reports. This is the operation that most users know about databases.
- *Database Maintenance* is used to add, delete, update, and correct the data in a database.
- *Application Development* is used to develop prototypes of data entry screens, queries, forms, reports, tables, and labels for a prototyped application. It also refers to the use of 4th Generation Language or application generators to develop or generate program code.

12.5. Data Management

A database must manage the data stored in it. This management includes both organization and storage.

The organization of the actual data in a database depends on the database model. In a relational model, data are organized as tables with different tables representing different entities or relations among a set of entities. The storage of data deals with the storage of these database tables on disks. The common ways for achieving this is to use files. Sequential, indexed, and hash files are all used in this purpose with different file structures providing different access performance and convenience.

12.6. Data Mining

One often has to know what to look for before querying a database. This type of "pinpointing" access does not make full use of the vast amount of information stored in the database, and in fact reduces the database into a collection of discrete records. To take full advantage of a database, one can perform statistical analysis and pattern discovery on the content of a database using a technique called *data mining.* Such operations can be used to support a number of business activities that include, but are not limited to, marketing, fraud detection, and trend analysis.

Numerous ways for performing data mining have been invented in the past decade and include such common techniques as class description, class discrimination, cluster analysis, association analysis, and outlier analysis.

13. Network Communication Basics

[8*, c12]

A computer network connects a collection of computers and allows users of different computers to share resources with other users. A network facilitates the communications between all the connected computers and may give the illusion of a single, omnipresent computer. Every computer or device connected to a network is called a *network node.*

A number of computing paradigms have emerged to benefit from the functions and capabilities provided by computer networks. These paradigms include distributed computing, grid computing, Internet computing, and cloud computing.

13.1. Types of Network

Computer networks are not all the same and may be classified according to a wide variety of characteristics, including the network's connection method, wired technologies, wireless technologies, scale, network topology, functions, and speed. But the classification that is familiar to most is based on the scale of networking.

- *Personal Area Network/Home Network* is a computer network used for communication among computer(s) and different information technological devices close to one person. The devices connected to such a network may include PCs, faxes, PDAs, and TVs. This is the base on which the Internet of Things is built.
- *Local Area Network* (LAN) connects computers and devices in a limited geographical area, such as a school campus, computer laboratory, office building, or closely positioned group of buildings.
- *Campus Network* is a computer network made up of an interconnection of local area networks (LANs) within a limited geographical area.
- *Wide area network* (WAN) is a computer network that covers a large geographic area, such as a city or country or even across intercontinental distances. A WAN limited to a city is sometimes called a Metropolitan Area Network.
- *Internet* is the global network that connects computers located in many (perhaps all) countries.

Other classifications may divide networks into control networks, storage networks, virtual private networks (VPN), wireless networks, point-to-point networks, and Internet of Things.

13.2. Basic Network Components

All networks are made up of the same basic hardware components, including computers, network

interface cards (NICs), bridges, hubs, switches, and routers. All these components are called *nodes* in the jargon of networking. Each component performs a distinctive function that is essential for the packaging, connection, transmission, amplification, controlling, unpacking, and interpretation of the data. For example, a repeater amplifies the signals, a switch performs many-to-many connections, a hub performs one-to-many connections, an interface card is attached to the computer and performs data packing and transmission, a bridge connects one network with another, and a router is a computer itself and performs data analysis and flow control to regulate the data from the network.

The functions performed by various network components correspond to the functions specified by one or more levels of the seven-layer Open Systems Interconnect (OSI) networking model, which is discussed below.

13.3. Networking Protocols and Standards

Computers communicate with each other using protocols, which specify the format and regulations used to pack and un-pack data. To facilitate easier communication and better structure, network protocols are divided into different layers with each layer dealing with one aspect of the communication. For example, the physical layers deal with the physical connection between the parties that are to communicate, the data link layer deals with the raw data transmission and flow control, and the network layer deals with the packing and un-packing of data into a particular format that is understandable by the relevant parties. The most commonly used OSI networking model organizes network protocols into seven layers, as depicted in Figure 13.5.

One thing to note is that not all network protocols implement all layers of the OSI model. For example, the TCP/IP protocol implements neither the presentation layer nor the session layer.

There can be more than one protocol for each layer. For example, UDP and TCP both work on the transport layer above IP's network layer, providing best-effort, unreliable transport (UDP) vs. reliable transport function (TCP). Physical layer protocols include token ring, Ethernet, fast Ethernet, gigabit Ethernet, and wireless Ethernet. Data link layer protocols include frame-relay, asynchronous transfer mode (ATM), and Point-to-Point Protocol (PPP). Application layer protocols include Fibre channel, Small Computer System Interface (SCSI), and Bluetooth. For each layer or even each individual protocol, there may be standards established by national or international organizations to guide the design and development of the corresponding protocols.

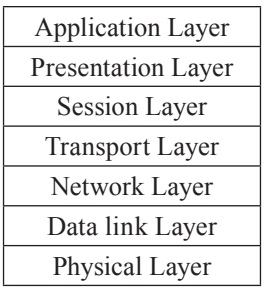

Figure 13.5. The Seven-Layer OSI Networking Model

13.4. The Internet

The Internet is a global system of interconnected governmental, academic, corporate, public, and private computer networks. In the public domain access to the internet is through organizations known as internet service providers (ISP). The ISP maintains one or more switching centers called a point of presence, which actually connects the users to the Internet.

13.5. Internet of Things

The Internet of Things refers to the networking of everyday objects—such as cars, cell phones, PDAs, TVs, refrigerators, and even buildings—using wired or wireless networking technologies. The function and purpose of *Internet of Things* is to interconnect all things to facilitate autonomous and better living. Technologies used in the Internet of Things include RFID, wireless and wired networking, sensor technology, and much software of course. As the paradigm of Internet of Things is still taking shape, much work is needed for Internet of Things to gain wide spread acceptance.

13.6. Virtual Private Network (VPN)

A virtual private network is a preplanned virtual connection between nodes in a LAN/WAN or on the internet. It allows the network administrator to separate network traffic into user groups that have a common affinity for each other such as all users in the same organization, or workgroup. This circuit type may improve performance and security between nodes and allows for easier maintenance of circuits when troubleshooting.

14. Parallel and Distributed Computing

[8*, c9]

Parallel computing is a computing paradigm that emerges with the development of multi-functional units within a computer. The main objective of parallel computing is to execute several tasks simultaneously on different functional units and thus improve throughput or response or both. Distributed computing, on the other hand, is a computing paradigm that emerges with the development of computer networks. Its main objective is to either make use of multiple computers in the network to accomplish things otherwise not possible within a single computer or improve computation efficiency by harnessing the power of multiple computers.

14.1. Parallel and Distributed Computing Overview

Traditionally, parallel computing investigates ways to maximize concurrency (the simultaneous execution of multiple tasks) within the boundary of a computer. Distributed computing studies distributed systems, which consists of multiple *autonomous* computers that communicate through a computer network. Alternatively, distributed computing can also refer to the use of distributed systems to solve computational or transactional problems. In the former definition, distributed computing investigates the protocols, mechanisms, and strategies that provide the foundation for distributed computation; in the latter definition, distributed computing studies the ways of dividing a problem into many tasks and assigning such tasks to various computers involved in the computation.

Fundamentally, distributed computing is another form of parallel computing, albeit on a grander scale. In distributed computing, the functional units are not ALU, FPU, or separate cores, but individual computers. For this reason, some people regard distributed computing as being the same as parallel computing. Because both distributed and parallel computing involve some form of concurrency, they are both also called concurrent computing.

14.2. Difference between Parallel and Distributed Computing

Though parallel and distributed computing resemble each other on the surface, there is a subtle but real distinction between them: parallel computing does not necessarily refer to the execution of programs on different computers— instead, they can be run on different processors within a single computer. In fact, consensus among computing professionals limits the scope of parallel computing to the case where a shared memory is used by all processors involved in the computing, while distributed computing refers to computations where private memory exists for each processor involved in the computations.

Another subtle difference between parallel and distributed computing is that parallel computing necessitates concurrent execution of several tasks while distributed computing does not have this necessity.

Based on the above discussion, it is possible to classify concurrent systems as being "parallel" or "distributed" based on the existence or nonexistence of shared memory among all the processor: parallel computing deals with computations within a single computer; distributed computing deals with computations within a set of computers. According to this view, multicore computing is a form of parallel computing.

14.3. Parallel and Distributed Computing Models

Since multiple computers/processors/cores are involved in distributed/parallel computing, some coordination among the involved parties is necessary to ensure correct behavior of the system.

Different ways of coordination give rise to different computing models. The most common models in this regard are the shared memory (parallel) model and the message-passing (distributed) model.

In a *shared memory (parallel)* model, all computers have access to a shared central memory where local caches are used to speed up the processing power. These caches use a protocol to insure the localized data is fresh and up to date, typically the MESI protocol. The algorithm designer chooses the program for execution by each computer. Access to the central memory can be synchronous or asynchronous, and must be coordinated such that coherency is maintained. Different access models have been invented for such a purpose.

In a *message-passing (distributed)* model, all computers run some programs that collectively achieve some purpose. The system must work correctly regardless of the structure of the network. This model can be further classified into client-server (C/S), browser-server (B/S), and n-tier models. In the C/S model, the server provides services and the client requests services from the server. In the B/S model, the server provides services and the client is the browser. In the n-tier model, each tier (i.e. layer) provides services to the tier immediately above it and requests services from the tier immediately below it. In fact, the n-tier model can be seen as a chain of client-server models. Often, the tiers between the bottommost tier and the topmost tier are called *middleware,* which is a distinct subject of study in its own right.

14.4. Main Issues in Distributed Computing

Coordination among all the components in a distributed computing environment is often complex and time-consuming. As the number of cores/CPUs/computers increases, the complexity of distributed computing also increases. Among the many issues faced, memory coherency and consensus among all computers are the most difficult ones. Many computation paradigms have been invented to solve these problems and are the main discussion issues in distributed/parallel computing.

15. Basic User Human Factors

[3*, c8] [9*, c5]

Software is developed to meet human desires or needs. Thus, all software design and development must take into consideration human-user factors such as how people use software, how people view software, and what humans expect from software. There are numerous factors in the human-machine interaction, and ISO 9241 document series define all the detailed standards of such interactions.[10] But the basic human-user factors considered here include input/output, the handling of error messages, and the robustness of the software in general.

15.1. Input and Output

Input and output are the interfaces between users and software. Software is useless without input and output. Humans design software to process some input and produce desirable output. All software engineers must consider input and output as an integral part of the software product they engineer or develop. Issues considered for input include (but are not limited to):

- What input is required?
- How is the input passed from users to computers?
- What is the most convenient way for users to enter input?
- What format does the computer require of the input data?

The designer should request the minimum data from human input, only when the data is not already stored in the system. The designer should format and edit the data at the time of entry to reduce errors arising from incorrect or malicious data entry.

For output, we need to consider what the users wish to see:

- In what format would users like to see output?
- What is the most pleasing way to display output?

If the party interacting with the software isn't human but another software or computer or control system, then we need to consider the input/output type and format that the software should produce to ensure proper data exchange between systems.

There are many rules of thumb for developers to follow to produce good input/output for a software. These rules of thumb include simple and natural dialogue, speaking users' language, minimizing user memory load, consistency, minimal surprise, conformance to standards (whether agreed to or not: e.g., automobiles have a standard interface for accelerator, brake, steering).

15.2. Error Messages

It is understandable that most software contains faults and fails from time to time. But users should be notified if there is anything that impedes the smooth execution of the program. Nothing is more frustrating than an unexpected termination or behavioral deviation of software without any warning or explanation. To be user friendly, the software should report all error conditions to the users or upper-level applications so that some measure can be taken to rectify the situation or to exit gracefully. There are several guidelines that define what constitutes a good error message: error messages should be clear, to the point, and timely.

First, error messages should clearly explain what is happening so that users know what is going on in the software. Second, error messages should pinpoint the cause of the error, if at all possible, so that proper actions can be taken. Third, error messages should be displayed right when the error condition occurs. According to Jakob Nielsen, "Good error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution" [9*]. Fourth, error messages should not overload the users with too much information and cause them to ignore the messages all together.

However, messages relating to security access errors should not provide extra information that would help unauthorized persons break in.

15.3. Software Robustness

Software robustness refers to the ability of software to tolerate erroneous inputs. Software is said to be robust if it continues to function even when erroneous inputs are given. Thus, it is unacceptable for software to simply crash when encountering an input problem as this may cause unexpected consequences, such as the loss of valuable data. Software that exhibits such behavior is considered to lack robustness.

Nielsen gives a simpler description of software robustness: "The software should have a low error rate, so that users make few errors during the use of the system and so that if they do make errors they can easily recover from them. Further, catastrophic errors must not occur" [9*].

There are many ways to evaluate the robustness of software and just as many ways to make software more robust. For example, to improve robustness, one should always check the validity of the inputs and return values before progressing further; one should always throw an exception when something unexpected occurs, and one should never quit a program without first giving users/applications a chance to correct the condition.

16. Basic Developer Human Factors

[3*, c31–32]

Developer human factors refer to the considerations of human factors taken when developing software. Software is developed by humans, read by humans, and maintained by humans. If anything is wrong, humans are responsible for correcting those wrongs. Thus, it is essential to write software in a way that is easily understandable by humans or, at the very least, by other software developers. A program that is easy to read and understand exhibits readability.

The means to ensure that software meet this objective are numerous and range from proper architecture at the macro level to the particular coding style and variable usage at the micro level. But the two prominent factors are *structure* (or program layouts) and *comments* (documentation).

16.1. Structure

Well-structured programs are easier to understand and modify. If a program is poorly structured, then no amount of explanation or comments is sufficient to make it understandable. The ways to organize a program are numerous and range from the proper use of white space, indentation, and parentheses to nice arrangements of groupings, blank lines, and braces. Whatever style one chooses, it should be consistent across the entire program.

16.2. Comments

To most people, programming is coding. These people do not realize that programming also includes writing comments and that comments are an integral part of programming. True, comments are not used by the computer and certainly do not constitute final instructions for the computer, but they improve the readability of the programs by explaining the meaning and logic of the statements or sections of code. It should be remembered that programs are not only meant for computers, they are also read, written, and modified by humans.

The types of comments include repeat of the code, explanation of the code, marker of the code, summary of the code, description of the code's intent, and information that cannot possibly be expressed by the code itself. Some comments are good, some are not. The good ones are those that explain the intent of the code and justify why this code looks the way it does. The bad ones are repeat of the code and stating irrelevant information. The best comments are self-documenting code. If the code is written in such a clear and precise manner that its meaning is self-proclaimed, then no comment is needed. But this is easier said than done. Most programs are not self-explanatory and are often hard to read and understand if no comments are given.

Here are some general guidelines for writing good comments:

- Comments should be consistent across the entire program.
- Each function should be associated with comments that explain the purpose of the function and its role in the overall program.
- Within a function, comments should be given for each logical section of coding to explain the meaning and purpose (intention) of the section.
- Comments should stipulate what freedom does (or does not) the maintaining programmers have with respect to making changes to that code.
- Comments are seldom required for individual statements. If a statement needs comments, one should reconsider the statement.

17. Secure Software Development and Maintenance

[11*, c29]

Due to increasing malicious activities targeted at computer systems, security has become a significant issue in the development of software. In addition to the usual correctness and reliability, software developers must also pay attention to the security of the software they develop. Secure software development builds security in software by following a set of established and/or recommended rules and practices in software development. Secure software maintenance complements secure software development by ensuring the no security problems are introduced during software maintenance.

A generally accepted view concerning software security is that it is much better to design security into software than to patch it in after software is developed. To design security into software, one must take into consideration every stage of the software development lifecycle. In particular, secure software development involves *software requirements security*, software *design security*, *software construction security,* and *software testing security*. In addition, security must also be taken into consideration when performing software maintenance as security faults and loopholes can be and often are introduced during maintenance.

17.1. Software Requirements Security

Software requirements security deals with the clarification and specification of security policy and objectives into software requirements, which

lays the foundation for security considerations in the software development. Factors to consider in this phase include software requirements and threats/risks. The former refers to the specific functions that are required for the sake of security; the latter refers to the possible ways that the security of software is threatened.

17.2. Software Design Security

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.

17.3. Software Construction Security

Software construction security concerns the question of how to write actual programming code for specific situations such that security considerations are taken care of. The term "Software Construction Security" could mean different things for different people. It can mean the way a specific function is coded, such that the coding itself is secure, or it can mean the coding of security into software.

Most people entangle the two together without distinction. One reason for such entanglement is that it is not clear how one can make sure that a specific coding is secure. For example, in C programming language, the expression of i<<1 (shift the binary representation of i's value to the left by one bit) and 2*i (multiply the value of variable i by constant 2) mean the same thing semantically, but do they have the same security ramification? The answer could be different for different combinations of ISAs and compilers. Due to this lack of understanding, software construction security—in its current state of existence—mostly refers to the second aspect mentioned above: the coding of security into software.

Coding of security into 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 should perform only those 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 that attackers will try to invalidate.
- Ensure that the program does not share objects in memory with any other program.
- The error status of every function must be checked. Do not try to recover unless neither the cause of the error 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.

17.4. Software Testing Security

Software testing security determines that software protects data and maintains security specification as given. For more information, please refer to the Software Testing KA.

17.5. Build Security into Software Engineering Process

Software is only as secure as its development process goes. To ensure the security of software, security must be built into the software engineering process. One trend that emerges in this regard is the Secure Development Lifecycle (SDL) concept, which is a classical spiral model that takes a holistic view of security from the perspective of software lifecycle and ensures that security is inherent in software design and development, not an afterthought later in production. The SDL process is claimed to reduce software maintenance costs and increase reliability of software concerning software security related faults.

17.6. Software Security Guidelines

Although there are no bulletproof ways for secure software development, some general guidelines do exist that can be used to aid such effort. These

guidelines span every phase of the software development lifecycle. Some reputable guidelines are published by the Computer Emergency Response Team (CERT) and below are its top 10 software security practices (the details can be found in [12]:

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.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	Voland 2003 [2*]	McConnell 2004 [3*]	Brookshear 2008 [4*]	Horowitz et al. 2007 [5*]	Sommerville 2011 [6*]	Null and Lobur 2006 [8*]	Nielsen 1993 [9*]	Bishop 2002 [11*]
1. Problem Solving Techniques	s3.2, s4.2							
1.1. Definition of Problem Solving	s3.2							
1.2. Formulating the Real Problem	s3.2							
1.3. Analyze the Problem	s3.2							
1.4. Design a Solution Search Strategy	s4.2							
1.5. Problem Solving Using Programs		c5						
2. Abstraction		s5.2–5.4						
2.1. Levels of Abstraction		s5.2–5.3						
2.2. Encapsulation		s5.3						
2.3. Hierarchy		s5.2						
3. Programming Fundamentals		c6–19						
3.1. The Programming Process		c6–c19						
3.2. Programming Paradigms		c6–c19						
3.3. Defensive Programming		c8						
4. Programming Language Basics			c6					
4.1. Programming Language Overview			s6.1					
4.2. Syntax and Semantics of Programming Language			s6.2					

	Voland 2003 [2*]	McConnell 2004 [3*]	Brookshear 2008 [4*]	Horowitz et al. 2007 [5*]	Sommerville 2011 [6*]	Null and Lobur 2006 [8*]	Nielsen 1993 [9*]	Bishop 2002 [11*]
4.3. Low Level Programming Language			s6.5–6.7					
4.4. High Level Programing Language			s6.5–6.7					
4.5. Declarative vs. Imperative Programming Language			s6.5–6.7					
5. Debugging Tools and Techniques		c23						
5.1. Types of Errors		s23.1						
5.2. Debugging Techniques:		s23.2						
5.3. Debugging Tools		s23.5						
6. Data Structure and Representation				s2.1–2.6				
6.1. Data Structure Overview				s2.1–2.6				
6.2. Types of Data Structure				s2.1–2.6				
6.3. Operations on Data Structures				s2.1–2.6				
7. Algorithms and Complexity				s1.1–1.3, s3.3–3.6, s4.1–4.8, s5.1–5.7, s6.1–6.3, s7.1–7.6, s11.1, s12.1				

	Voland 2003 [2*]	McConnell 2004 [3*]	Brookshear 2008 [4*]	Horowitz et al. 2007 [5*]	Sommerville 2011 [6*]	Null and Lobur 2006 [8*]	Nielsen 1993 [9*]	Bishop 2002 [11*]
7.1. Overview of Algorithms				s1.1–1.2				
7.2. Attributes of Algorithms				s1.3				
7.3. Algorithmic Analysis				s1.3				
7.4. Algorithmic Design Strategies				s3.3–3.6, s4.1–4.8, s5.1–5.7, s6.1–6.3, s7.1–7.6, s11.1, s12.1				
7.5. Algorithmic Analysis Strategies				s3.3–3.6, s4.1–4.8, s5.1–5.7, s6.1–6.3, s7.1–7.6, s11.1, s12.1				
8. Basic Concept of a System					c10			
8.1. Emergent System Properties					s10.1			
8.2. System Engineering					s10.2			
8.3. Overview of a Computer System								

	Voland 2003 [2*]	McConnell 2004 [3*]	Brookshear 2008 [4*]	Horowitz et al. 2007 [5*]	Sommerville 2011 [6*]	Null and Lobur 2006 [8*]	Nielsen 1993 [9*]	Bishop 2002 [11*]
9. Computer Organization						c1–4		
9.1. Computer Organization Overview						s1.1–1.2		
9.2. Digital Systems						c3		
9.3. Digital Logic						c3		
9.4. Computer Expression of Data						c2		
9.5. The Central Processing Unit (CPU)						s4.1–4.2		
9.6. Memory System Organization						s4.6		
9.7. Input and Output (I/O)						s4.5		
10. Compiler Basics			s6.4			s8.4		
10.1. Compiler Overview						s8.4		
10.2. Interpretation and Compilation						s8.4		
10.3. The Compilation Process			s6.4			s8.4		
11. Operating Systems Basics			c3					
11.1. Operating Systems Overview			s3.2					
11.2. Tasks of Operating System			s3.3					
11.3. Operating System Abstractions			s3.2					
11.4. Operating Systems Classification			s3.1					

	Voland 2003 [2*]	McConnell 2004 [3*]	Brookshear 2008 [4*]	Horowitz et al. 2007 [5*]	Sommerville 2011 [6*]	Null and Lobur 2006 [8*]	Nielsen 1993 [9*]	Bishop 2002 [11*]
12. Database Basics and Data Management			c9					
12.1. Entity and Schema			s9.1					
12.2. Database Management Systems (DBMS)			s9.1					
12.3. Database Query Language			s9.2					
12.4. Tasks of DBMS Packages			s9.2					
12.5. Data Management			s9.5					
12.6. Data Mining			s9.6					
13. Network Communication Basics						c12		
13.1. Types of Network						s12.2–12.3		
13.2. Basic Network Components						s12.6		
13.3. Networking Protocols and Standards						s12.4–12.5		
13.4. The Internet								
13.5. Internet of Things						s12.8		
13.6. Virtual Private Network								
14. Parallel and Distributed Computing						c9		
14.1. Parallel and Distributed Computing Overview						s9.4.1–9.4.3		

	Voland 2003 [2*]	McConnell 2004 [3*]	Brookshear 2008 [4*]	Horowitz et al. 2007 [5*]	Sommerville 2011 [6*]	Null and Lobur 2006 [8*]	Nielsen 1993 [9*]	Bishop 2002 [11*]
14.2. Differences between Parallel and Distributed Computing						s9.4.4–9.4.5		
14.3. Parallel and Distributed Computing Models						s9.4.4–9.4.5		
14.4. Main Issues in Distributed Computing								
15. Basic User Human Factors		c8					c5	
15.1. Input and Output							s5.1, s5.3	
15.2. Error Messages							s5.2, s5.8	
15.3. Software Robustness							s5.5–5.6	
16. Basic Developer Human Factors		c31–32						
16.1. Structure		c31						
16.2. Comments		c32						
17. Secure Software Development and Maintenance								c29
17.1. Two Aspects of Secure Coding								s29.1
17.2. Coding Security into Software								s29.4
17.3. Requirement Security								s29.2
17.4. Design Security								s29.3
17.5. Implementation Security								s29.5

REFERENCES

[1] Joint Task Force on Computing Curricula, IEEE Computer Society and Association for Computing Machinery, *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*, 2004; 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*, 10th ed., Addison-Wesley, 2008.

[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 24765:2010 Systems and Software Engineering—Vocabulary*, ISO/IEC/IEEE, 2010.

[8*] L. Null and J. Lobur, *The Essentials of Computer Organization and Architecture*, 2nd ed., Jones and Bartlett Publishers, 2006.

[9*] J. Nielsen, *Usability Engineering*, Morgan Kaufmann, 1993.

[10] ISO 9241-420:2011 Ergonomics of Human-System Interaction, ISO, 2011.

[11*] M. Bishop, *Computer Security: Art and Science*, Addison-Wesley, 2002.

[12] R.C. Seacord, *The CERT C Secure Coding Standard*, Addison-Wesley Professional, 2008.

CHAPTER 14

MATHEMATICAL FOUNDATIONS

INTRODUCTION

Software professionals live with programs. In a very simple language, one can program only for something that follows a well-understood, nonambiguous logic. The Mathematical Foundations knowledge area (KA) helps software engineers comprehend this logic, which in turn is translated into programming language 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 preciseness, 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, not only numbers and numeric equations are subject to preciseness. On the contrary, a software engineer needs to have a precise abstraction on a diverse application domain.

The *SWEBOK Guide*'s 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 here allow us to describe mathematical proofs to infer conclusively the absolute truth of certain concepts beyond the numbers. In short, you can write a program for a problem only if it follows some logic. The objective of this KA is to help you develop the skill to identify and describe such logic. The emphasis is on helping you understand the basic concepts rather than on challenging your arithmetic abilities.

BREAKDOWN OF TOPICS FOR MATHEMATICAL FOUNDATIONS

The breakdown of topics for the Mathematical Foundations KA is shown in Figure 14.1.

1. Set, Relations, Functions

[1*, c2]

Set. A set is a collection of objects, called elements of the set. 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 | 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, …} = the set of nonnegative integers.
Z = {…, −3, −2, −1, 0, 1, 2, 3, …} = 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.

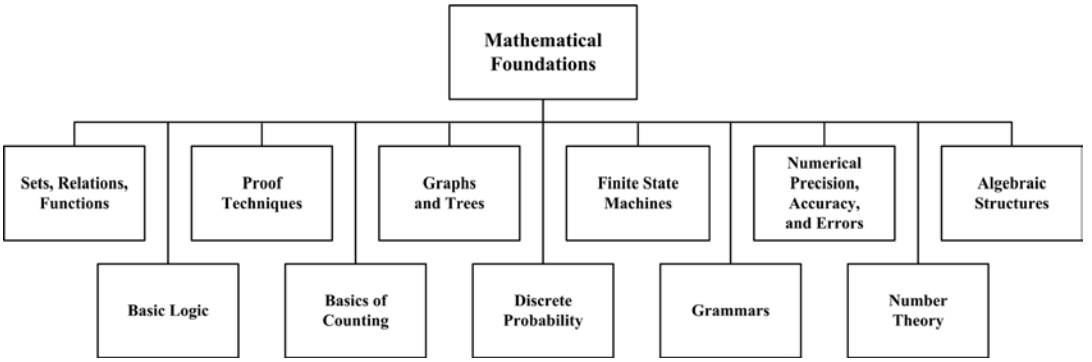

Figure 14.1. Breakdown of Topics for the Mathematical Foundations KA

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 ∈ U | 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 $\forall p\ (p \in X \rightarrow p \in Y)$.

For example, 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 $(X \subseteq Y) \wedge (X \neq Y)$.

For example, 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 $X \subseteq Y$.

For example, 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 ∅, 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 $\wp(X)$.

For example, if X = {a, b, c}, then $\wp(X)$ = {∅, {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.

For example, in Figure 14.2, the rectangle represents the universal set and the shaded region represents a set X.

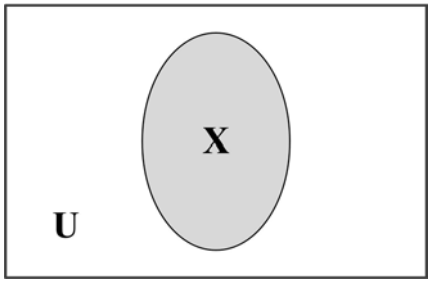

Figure 14.2. Venn Diagram for Set X

1.1. 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, $X \cap Y = \{p \mid (p \in X) \wedge (p \in Y)\}$.

As, for example, $\{1, 2, 3\} \cap \{3, 4, 6\} = \{3\}$

If $X \cap Y = f$, then the two sets X and Y are said to be a disjoint pair of sets.

A Venn diagram for set intersection is shown in Figure 14.3. The common portion of the two sets represents the set intersection.

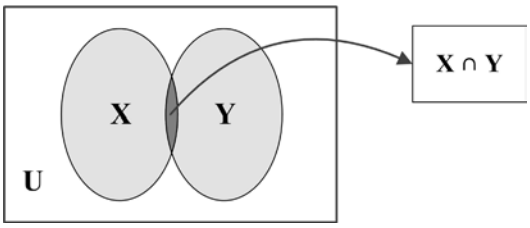

Figure 14.3. Intersection of Sets X and Y

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, $X \cup Y = \{p \mid (p \in X) \vee (p \in Y)\}$.

As, for example, $\{1, 2, 3\} \cup \{3, 4, 6\} = \{1, 2, 3, 4, 6\}$.

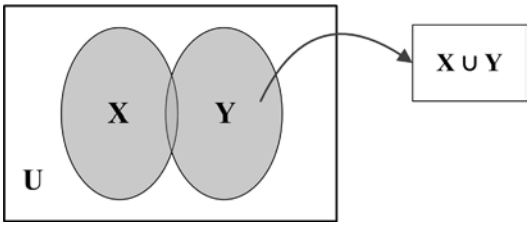

Figure 14.4. Union of Sets X and Y

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 14.4.

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, $X' = \{p \mid (p \in U) \wedge (p \notin X)\}$.

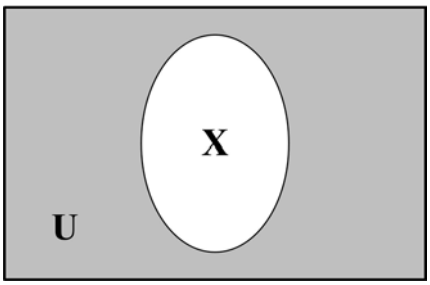

Figure 14.5. Venn Diagram for Complement Set of X

The shaded portion of the Venn diagram in Figure 14.5 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, $X - Y = \{p \mid (p \in X) \wedge (p \notin Y)\}$.

As, 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 14.6 using a Venn diagram.

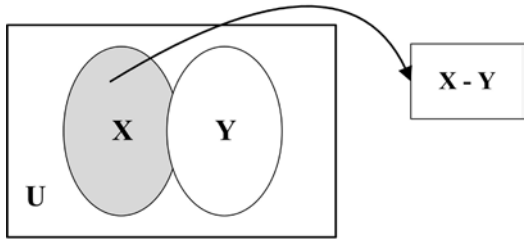

Figure 14.6. Venn Diagram for $X - Y$

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, $X \times Y = \{(p, q) \mid (p \in X) \wedge (q \in Y)\}$.

As for example, $\{a, b\} \times \{1, 2\} = \{(a, 1), (a, 2), (b, 1), (b, 2)\}$

1.2. 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:
   $X \cup Y = Y \cup X$ $X \cap Y = Y \cap X$

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:
   $X \cup \varnothing = X$ $X \cap U = X$

5. Complement Laws:
   $X \cup X' = U$ $X \cap X' = \varnothing$

6. Idempotent Laws:
   $X \cup X = X$ $X \cap X = X$

7. Bound Laws:
   $X \cup U = U$ $X \cap \varnothing = \varnothing$

8. Absorption Laws:
   $X \cup (X \cap Y) = X$ $X \cap (X \cup Y) = X$

9. De Morgan's Laws:
   $(X \cup Y)' = X' \cap Y'$ $(X \cap Y)' = X' \cup Y'$

1.3. Relation and Function

A relation is an association between two sets of information. For example, let's 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. In the example being considered, 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 up 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. Let's consider domain set X as a set of persons and let range set Y store their phone numbers. Assuming that a person may have more than one phone number, the relation being considered 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, let's 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 case of relation A, the domain is all the x-values, i.e., {3, 5, 6, 7}, and the range is all the 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 of 3.

In case of relation B, the domain is same as that for A, i.e., {3, 5, 6, 7}. However, the range is a single element {8}. This qualifies as an example of a function even if all the 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 a 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 the relation is not a function.

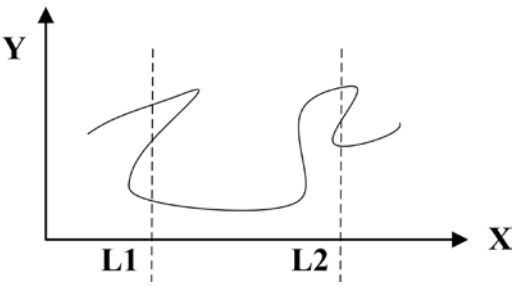

Figure 14.7. Vertical Line Test for Function

In this example, both lines L1 and L2 cut the graph for the relation thrice. This signifies that for the same x-value, there are three different y-values for each of case. Thus, the relation is not a function.

2. Basic Logic

[1*, c1]

2.1. Propositional Logic

A proposition is a statement that is either true or false, but not both. Let's 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 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 including negation ($\neg$ p), conjunction (p $\wedge$ q), disjunction (p $\vee$ q), exclusive or (p $\oplus$ q), and implication (p $\rightarrow$ q) are to be studied. 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:
$p \wedge T \equiv p \qquad p \vee F \equiv p$

Domination laws:
$p \vee T \equiv T \qquad p \wedge F \equiv F$

Idempotent laws:
$p \vee p \equiv p \qquad p \wedge p \equiv p$

Double negation law:
$\neg (\neg p) \equiv p$

Commutative laws:
$p \vee q \equiv q \vee p \qquad p \wedge q \equiv q \wedge p$

Associative laws:
$(p \vee q) \vee r \equiv p \vee (q \vee r)$
$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$

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:
$\neg (p \wedge q) \equiv \neg p \vee \neg q \qquad \neg (p \vee q) \equiv \neg p \wedge \neg q$

2.2. 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 too.

Predicates are often given a name, e.g., "Red" or simply "R" 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 of the color 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 Tiger(x) $\rightarrow$ 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 Tiger(x) $\rightarrow$ 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.

A variable is said to be a free variable if it is not bound to a 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 $\exists x\ (Cat(x) \wedge \forall x\ (Black(x)))$, x in Black(x) is universally quantified. The expression implies that cats exist and everything is black.

Propositional logic falls short in representing many assertions that are used in computer science and mathematics. 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 find out whether or not 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.

3. 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, the conjecture becomes a theorem. Many times conjectures are shown to be false and, hence, are not theorems.

3.1. Methods of Proving Theorems

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.

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.

Proof by Induction. Proof by induction is done in two phases. First, the proposition is established to be true for a base case—typically for the

positive integer 1. In the second phase, 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), $\forall n \in [1 \ldots \infty]$ is established if P(1) is true, and secondly, $\forall k \in [2 \ldots n]$ if P(k) $\rightarrow$ P(k + 1).

It may be noted here that, for a proof by mathematical 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 mathematical induction as a valid proof technique is beyond discussion of the current text. Let us prove the following proposition 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.

$$\therefore 1 + 3 + 5 + \ldots + (2k - 1) = k^2$$

Now, it's to be shown that P(k) $\rightarrow$ P(k + 1).

$$\begin{aligned} P(k+1) &= 1 + 3 + 5 + \ldots + (2k-1) + (2k+1) \\ &= P(k) + (2k+1) \\ &= k^2 + (2k+1) \text{ [using IH]} \\ &= k^2 + 2k + 1 \\ &= (k+1)^2 \end{aligned}$$

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.

4. 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 A1, A2, …. , An are disjoint sets, then $|A1 \cup A2 \cup \ldots \cup An| = |A1| + |A2| + \ldots + |An|$.

For example, 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 A1, A2, …, An are disjoint sets, then $|A1 \times A2 \times \ldots \times An| = |A1| * |A2| * \ldots * |An|$.

For example, 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 define $f(n) = 3 * f(n - 1)$ for all $n \in N$ 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 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, ${}^{n}C_{r} = n! / [r! * (n - r)!]$.

5. Graphs and Trees

[1*, c10, c11]

5.1. Graphs

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.

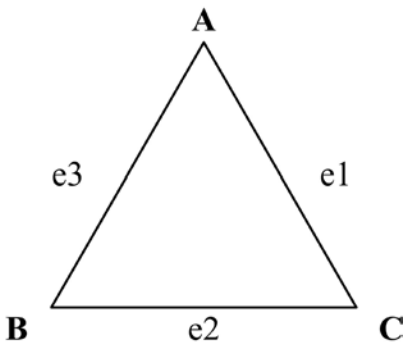

Figure 14.8. Example of a Graph

F is a function that maps the set of edges E to a set of ordered or unordered pairs of elements V. For example, in Figure 14.8, 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 14.8 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.

For example, in Figure 14.8, (e1, (A, C)) may be replaced by (e1, (C, A)) as the pair between vertices A and C is unordered. This holds good for the other two edges too.

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.

For example, in Figure 14.9, the edges e3 and e4 are both between A and B. Figure 14.9 is a multigraph where edges e3 and e4 are multiple edges.

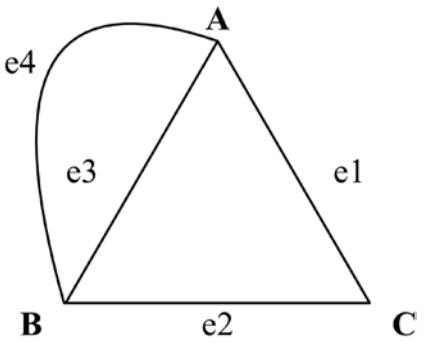

Figure 14.9. Example of a Multigraph

In a *pseudograph,* edges connecting a node to itself are allowed. Such edges are called loops.

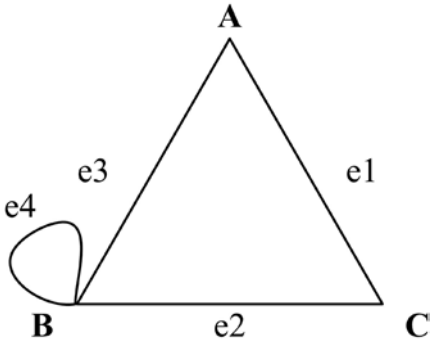

Figure 14.10. Example of a Pseudograph

For example, in Figure 14.10, the edge e4 both starts and ends at B. Figure 14.10 is a pseudograph in which e4 is a loop.

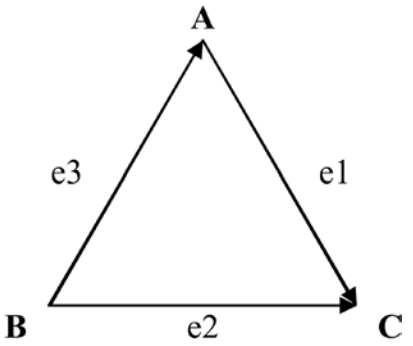

Figure 14.11. Example of a Directed Graph

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.

For example, in Figure 14.11, G = (V, E) where V = {A, B, C}, E = {e1, e2, e3}, and F = {(e1, (A, C)), (e2, (B, C)), (e3, (B, A))}.

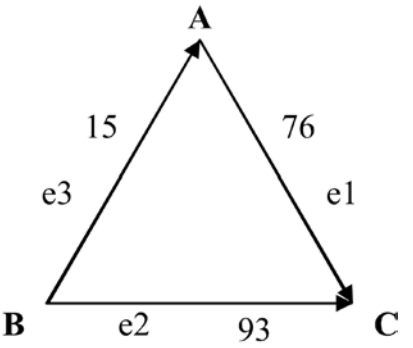

Figure 14.12. Example of a Weighted Graph

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.

For example, in Figure 14.12, the weights for the edges e1, e2, and e3 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 miles between these cities.

Let G = (V, E) be an undirected graph with edge set E. Then, for an edge e ∈ 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 ∈ 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.

It may be noted that, 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 $V = \{v_1, v_2, \ldots, v_n\}$ and $E = \{\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{n-1}, v_n\}, \{v_n, v_1\}\}$.

For example, Figure 14.13 illustrates two cycles of length 3 and 4.

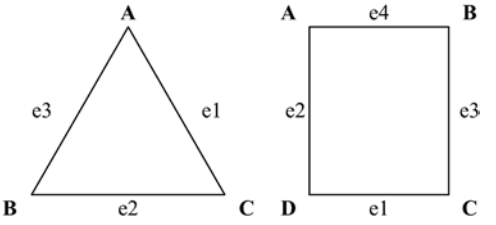

Figure 14.13. Example of Cycles C_3 and C_4

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 14.14. Adjacency Lists for Graphs in Figures 14.10 and 14.11

For example, Figure 14.14 illustrates the adjacency lists for the pseudograph in Figure 14.10 and the directed graph in Figure 14.11. As the out-degree of vertex C in Figure 14.11 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.

5.2. Trees

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.

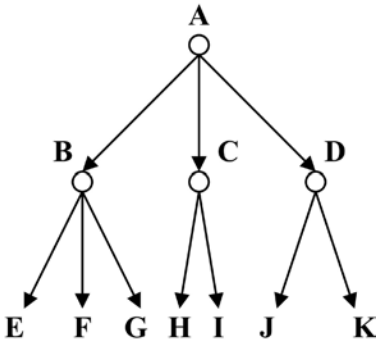

Figure 14.15. Example of a Tree

However, one should remember that a tree is strictly hierarchical in nature as compared to a graph, which is flat. In case of 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 14.15 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 nonroot 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.

For example, in Figure 14.15, 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.

For example, in Figure 14.15, node A or node B are examples of internal nodes.

The degree of a node in a tree is the same as its number of children.

For example, in Figure 14.15, 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.

For example, root node A is at level 0 in Figure 14.15. Nodes B, C, and D are at level 1. The remaining nodes in Figure 14.15 are all at level 2.

The height of a tree is the maximum of the levels of nodes in the tree.

For example, in Figure 14.15, 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.

For example, in Figure 14.15, nodes E through K are all leaf nodes with degree 0.

The ancestors or predecessors of a nonroot node X are all the nodes in the path from root to node X.

For example, in Figure 14.15, nodes A and D form the set of ancestors for J.

The successors or descendents of a node X are all the nodes that have X as its ancestor. For a tree with n nodes, all the remaining n − 1 nodes are successors of the root node.

For example, in Figure 14.15, 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.

For example, in Figure 14.15, 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 appears always 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 two 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.

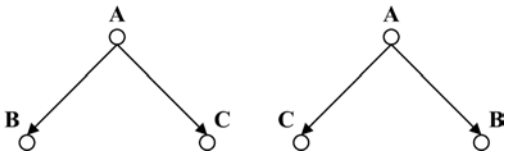

Figure 14.16. Examples of Binary Trees

For example, in Figure 14.16, the two binary trees are different as the positions of occurrences of the children of A are different in the two trees.

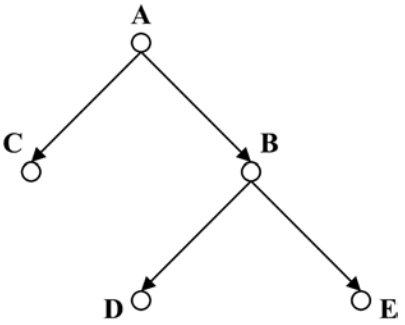

Figure 14.17. Example of a Full Binary Tree

According to [1*], a binary tree is called a full binary tree if every internal node has exactly two children.

For example, the binary tree in Figure 14.17 is a full binary tree, as both of the two 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*.

For example, both binary trees in Figure 14.18 are complete binary trees. The tree in Figure 14.18(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 up to capacity. In case the last level of a complete binary tree is not full, nodes occur from the leftmost positions available.

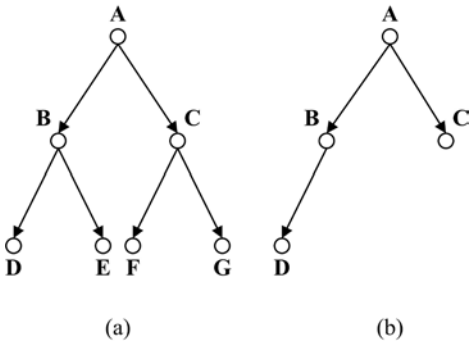

Figure 14.18. Example of Complete Binary Trees

Interestingly, following the definitions above, the tree in Figure 14.18(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 14.17 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 its leaf nodes occur at levels H or H − 1.

For example, all three binary trees in Figures 14.17 and 14.18 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 $H = \lceil \log_2 L \rceil$.

For example, this statement is true for the two trees in Figures 14.17 and 14.18(a) as both trees are full and balanced. However, the expression above does not match for the tree in Figure 14.18(b) as it 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:

$$\text{PreOrder}(T) = R, \text{PreOrder}(T_L), \text{PreOrder}(T_R) \quad \ldots \text{ eqn. 1}$$

The recursive process of finding the preorder traversal of the subtrees continues till 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.

$PostOrder(T) = PostOrder(T_L), PostOrder(T_R), R$ … eqn 2

$InOrder(T) = InOrder(T_L), R, InOrder(T_R)$ … eqn 3

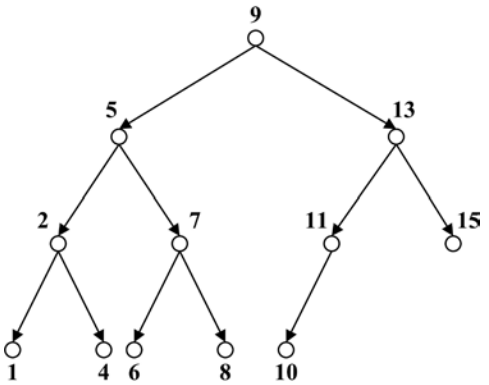

Figure 14.19. A Binary Search Tree

For example, the tree in Figure 14.19 is a binary search tree (BST). The preorder, postorder, and in-order traversal outputs for the 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.

6. Discrete Probability

[1*, c7]

Probability is the mathematical description of randomness. Basic definition of probability and randomness has been defined in section 4 of this KA. Here, let us 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 registered 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 a 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 contrary, for the same interval of 1 to 100, another random variable Y can be used to list all the 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:

$P(X = x)$ or, more simply, $P(x)$.

A probability distribution (density) function 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. $\Sigma P(x) = 1$

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 14.20. 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 $x_1, x_2, \ldots, x_n$ in a sample space S if p_k is the probability of outcome x_k, the mean of this probability would be $\mu = x_1p_1 + x_2p_2 + \ldots + x_np_n$.

For example, the mean of the probability density for the distribution in Figure 14.20 would be

$1 * (1/6) + 2 * (1/6) + 3 * (1/6) + 4 * (1/6) + 5 * (1/6) + 6 * (1/6)$
$= 21 * (1/6) = 3.5$

Here, the sample space refers to the set of all possible outcomes.

The variance s^2 of a discrete probability model is: $s^2 = (x_1 - \mu)^2p_1 + (x_2 - \mu)^2p_2 + \ldots + (x_k - \mu)^2p_k$. The *standard deviation*s is the square root of the variance.

For example, for the probability distribution in Figure 14.20, the variation σ^2 would be

$s^2 = [(1 - 3.5)^2 * (1/6) + (2 - 3.5)^2 * (1/6) + (3 - 3.5)^2 * (1/6) + (4 - 3.5)^2 * (1/6) + (5 - 3.5)^2 * (1/6) + (6 - 3.5)^2 * (1/6)]$
$= (6.25 + 2.25 + 0.25 + 0.5 + 2.25 + 6.25) * (1/6)$
$= 17.5 * (1/6)$
$= 2.90$

$\therefore$ standard deviation s =

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.

7. Finite State Machines

[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 (not infinite), 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 in a sequence and one at a time, 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 ∈ I on state S_k, the FSM makes a transition to state S_h following state transition function f and produces an output y ∈ O using the output function g.

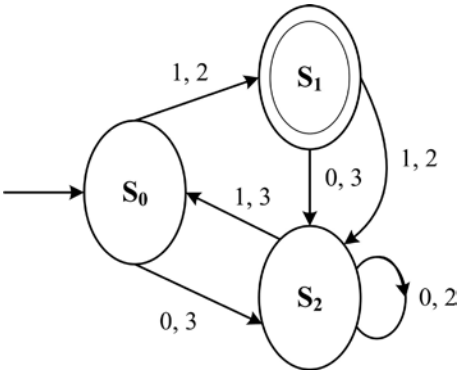

Figure 14.21. Example of an FSM

For example, Figure 14.21 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	
	0	1
S_0	S_2	S_1
S_1	S_2	S_2
S_2	S_2	S_0

(a)

Current State	Output		State	
	Input		Input	
	0	1	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)

Figure 14.22. Tabular Representation of an FSM

The state transition and output values for different inputs on different states may be represented using a state table. The state table for the FSM in Figure 14.21 is shown in Figure 14.22. Each pair against an input symbol represents the new state and the output symbol.

For example, Figures 14.22(a) and 14.22(b) are two alternate representations of the FSM in Figure 14.21.

8. Grammars

[1*, c13]

The grammar of a natural language tells us 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 these 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 of the language's legal sentences or examples of those sentences.

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 ⊆ V is a set of words called terminals.
- S ∈ N is a special word called the start symbol.
- P is the set of productions 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 → aA, S → b, A → 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 → aA; P2: S → b; P3: A → aa.

Applying the production rules in all possible ways, the following words may be generated from the start symbol.

S → aA	(using P1 on start symbol)
→ aaa	(using P3)
S → b	(using P2 on start symbol)

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.

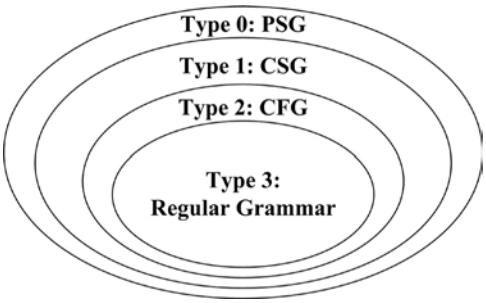

Figure 14.23. Chomsky Hierarchy of Grammars

As illustrated in Figure 14.23, 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 → a, then |b| < |a| or a = ∅.

A formal language is context-sensitive if a context-sensitive grammar generates it.

Context-Free Grammar: All fragments in the LHS are of length 1, i.e., if A → a, then |A| = 1 for all A ∈ 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 → a, then either a ∈ T, or a = cD, or a = Dc for c ∈ T, D ∈ 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 ∈ S, the set of alphabets, e, 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 ∈ S*|x matches G}.

For any a ∈ S, L(a) = a; L(e) = {ε}; L(0) = 0.
+ functions as an or, L(A + B) = L(A) ∪ L(B).
. creates a product structure, L(AB) = L(A) . L(B).
* denotes concatenation, L(A*) = $\{x_1x_2\ldots x_n \mid x_i \in L(A)$ and $n \geq 0\}$

For example, the regular expression (ab)* matches the set of strings: {e, ab, abab, ababab, abababab, …}.

For example, the regular expression (aa)* matches the set of strings on one letter *a* that have even length.

For example, the regular expression (aaa)* + (aaaaa)* matches the set of strings of length equal to a multiple of 3 or 5.

9. Numerical Precision, Accuracy, and Errors [2*, c2]

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.

A matter of fact is that all digital computers can only store finite numbers. In other words, there is no way that a computer can represent an infinitely large number—be it an integer, rational number, or any real or all complex numbers (see section 10, Number Theory). So the mathematics of approximation becomes very critical to handle all the numbers in the finite range that a computer can handle.

Each number in a computer is assigned a location or word, consisting of a specified number of binary digits or bits. A k bit word can store a total of $N = 2^k$ different numbers.

For example, a computer that uses 32 bit arithmetic can store a total of $N = 2^{32} \approx 4.3 \times 10^9$ different numbers, while another one that uses 64 bits can handle $N' = 2^{64} \approx 1.84 \times 10^{19}$ different numbers. The question is how to distribute these N numbers over the real line for maximum efficiency and accuracy in practical computations.

One evident choice is to distribute them evenly, leading to fixed-point arithmetic. In this system, the first bit in a word is used to represent a sign and the remaining bits are treated for integer values. This allows representation of the integers from $1 - \frac{1}{2}N$, i.e., $= 1 - 2^{k-1}$ to 1. As an approximating method, this is not good for noninteger numbers.

Another option 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 lying beyond the range must be represented by the largest (or largest negative) number that can be represented. This becomes a symbol for overflow. Overflow occurs when a computation produces a value larger than the maximum value in the range.

When processing speed is a significant bottleneck, the use of the fixed-point representations is an attractive and faster alternative to the more cumbersome floating-point arithmetic most commonly used in practice.

Let's define a couple of very important terms: *accuracy* and *precision* as associated with 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 physical substance 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 to 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.

10. 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.

10.1. Divisibility

Let's start this section with a brief description of each of the above types of numbers, starting with the natural 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 of the natural numbers in it 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 = {…, −3, −2, −1, 0, 1, 2, 3, …}.

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..., (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., PI 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*, 5*i*, 3.2*i*, −2.6*i,* etc.

Complex Numbers. A complex number is a combination of a real number and an imaginary number in the form a + b*i*. The real part is a, and b is called the imaginary part. The common mathematical symbol for the set of all complex numbers is **C.**

For example, 2 + 3*i*, 3−5*i*, 7.3 + 0*i*, and 0 + 5*i*.

Consider the last two examples:

7.3 + 0*i* is the same as the real number 7.3. Thus, all real numbers are complex numbers with zero for the imaginary part.

Similarly, 0 + 5*i* is just the imaginary number 5*i*. Thus, all imaginary numbers are complex numbers with zero for the real part.

Elementary number theory involves divisibility among integers. Let a, b ∈ Z with a ≠ 0.The expression a|b, i.e., *a divides b* if ∃c ∈ 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* ∈ Z with d > 1. Then *a mod 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 mod d)* by: $a - d * \lfloor a/d \rfloor$, where $\lfloor a/d \rfloor$ represents the floor of the real number.

Let $Z^+ = \{n \in Z \mid n > 0\}$ and *a, b* ∈ Z, m ∈ Z^+, then a is congruent to *b modulo m*, written as $a \equiv b \pmod m$, if and only if $m \mid a-b$.

Alternately, *a* is congruent to *b modulo m* if and only if *(a−b) mod m = 0*.

10.2. Prime Number, GCD

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 $p > 1 \wedge \exists \neg a, b \in N: a > 1, b > 1, a * b = p$.

The only positive factors of a prime p are 1 and p itself. For example, 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.

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.

For example, 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 $X = \{i_1, i_2, \ldots\}$ is relatively prime if all possible pairs $i_h, i_k, h \neq k$ drawn from the set X are relatively prime.

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 a number of axioms, including none.

For example, group, monoid, ring, and lattice are examples of algebraic structures. Each of these is defined in this section.

11.1. Group

A set S closed under a binary operation • 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 rule (of inverse) for our operation is violated. However, the set of natural number has some structure.

Sets with an associative operation (the first condition above) are called semigroups; if they also have an identity element (the second condition), then they are called monoids.

Our set of natural numbers under addition is then 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 • 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.

For example, 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 $G = \{a^n$ 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. Rings

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, +, •), where (S, +) is an Abelian group, (S, •) is a semigroup, and • is distributive over +; i.e., " a, b, c ∈ S, the equation $a \bullet (b + c) = (a \bullet b) + (a \bullet c)$ holds. Further, if • is commutative, then the ring is said to be commutative. If there is an identity element for the • operation, then the ring is said to have an identity.

For 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.

Let's 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 $a \neq 0$.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	Rosen 2011 [1*]	Cheney and Kincaid 2007 [2*]
1. Sets, Relations, Functions	c2	
2. Basic Logic	c1	
3. Proof Techniques	c1	
4. Basic Counting	c6	
5. Graphs and Trees	c10, c11	
6. Discrete Probability	c7	
7. Finite State Machines	c13	
8. Grammars	c13	
9. Numerical Precision, Accuracy, and Errors		c2
10. Number Theory	c4	
11. Algebraic Structures		

REFERENCES

[1*] K. Rosen, *Discrete Mathematics and Its Applications*, 7th ed., McGraw-Hill, 2011.

[2*] E.W. Cheney and D.R. Kincaid, *Numerical Mathematics and Computing*, 6th ed., Brooks/Cole, 2007.

ACKNOWLEDGMENTS

The author thankfully acknowledges the contribution of Prof. Arun Kumar Chatterjee, Ex-Head, Department of Mathematics, Manipur University, India, and Prof. Devadatta Sinha, Ex-Head, Department of Computer Science and Engineering, University of Calcutta, India, in preparing this chapter on Mathematical Foundations.

CHAPTER 15

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]. This chapter outlines some of the engineering foundational skills and techniques that are useful for a software engineer. The focus is on topics that support other KAs while minimizing duplication of subjects covered elsewhere in this document.

As the theory and practice of software engineering matures, it is increasingly apparent that software engineering is an engineering discipline that is based on knowledge and skills common to all engineering disciplines. This Engineering Foundations knowledge area (KA) is concerned with the engineering foundations that apply to software engineering and other engineering disciplines. Topics in this KA include empirical methods and experimental techniques; statistical analysis; measurement; engineering design; modeling, prototyping, and simulation; standards; and root cause analysis. Application of this knowledge, as appropriate, will allow software engineers to develop and maintain software more efficiently and effectively. Completing their engineering work efficiently and effectively is a goal of all engineers in all engineering disciplines.

BREAKDOWN OF TOPICS FOR ENGINEERING FOUNDATIONS

The breakdown of topics for the Engineering Foundations KA is shown in Figure 15.1.

1. Empirical Methods and Experimental Techniques

[2*, c1]

An engineering method for problem solving involves proposing solutions or models of solutions and then conducting experiments or tests to study the proposed solutions or models. Thus, engineers must understand how to create an experiment and then analyze the results of the experiment in order to evaluate the proposed solution. Empirical methods and experimental techniques help the engineer to describe and understand variability in their observations, to identify the sources of variability, and to 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 below.

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.

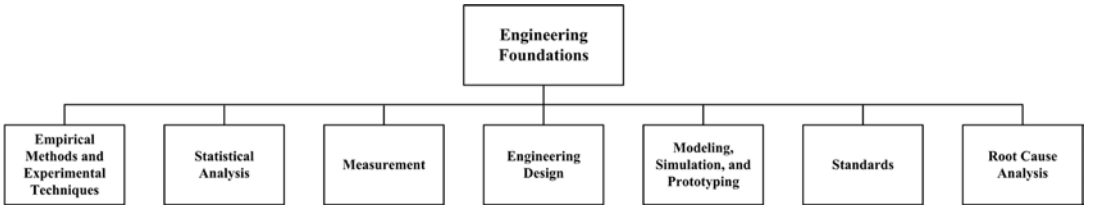

Figure 15.1. Breakdown of Topics for the Engineering Foundations KA

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.

1.2. Observational Study

An observational or case study is an empirical inquiry that makes observations of processes or phenomena within a real-life 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.

1.3. 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 in an attempt to find a relationship between variables, to predict future events, or to identify trends. The quality of the analysis results will depend on the quality of the information contained in the archived data. Historical data may be incomplete, inconsistently measured, or incorrect.

2. Statistical Analysis

[2*, c9s1, c2s1] [3*, c10s3]

In order to carry out their responsibilities, 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 on the basis of 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 below.

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, when a software product company wants 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, consider the case of studying the perceived usability of a software product. In this case, the set of all possible users forms the population.

While defining the population, care must be exercised to understand the study and target population. There are cases when the population 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 so as to ensure that the draws are independent, and the rules of drawing the 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 then 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 [2*, c3s6].
- Poisson distribution: used to model the count of occurrence of some event over time or space [2*, c3s9].
- Normal distribution: used to model continuous random variables or discrete random variables by taking a very large number of values [2*, c4s6].

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, called the 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, called the 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 pdf

or pmf is known, the chances of the random variable taking certain set of values may be computed theoretically.

Concept of estimation [2*, c6s2, 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 to note 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 alternative 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 [2*, 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 on the basis of some 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 [2*, 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. In this case, the hypothesis is that the rate of occurrence of defects has reduced. In tests of hypotheses, we decide—on the basis of 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).

In order to test some hypothesis, we first compute some statistic. Along with the computation of the statistic, a region is defined such that in case 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 on the basis of 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 on the basis of 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 = a)
H_0 is false	Type II error (probability = b)	OK

In test of hypotheses, we aim at maximizing the power of the test (the value of 1−b) while ensuring that the probability of a type I error (the value of a) is maintained within a particular value—typically 5 percent.

It is to be noted 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. Concepts of Correlation and Regression

[2*, 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 have to 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 in the following paragraphs.

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.

It is important to 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. Measurement

[4*, c3s1, c3s2] [5*, c4s4] [6*, c7s5] [7*, p442–447]

Knowing what to measure and which measurement method to use is critical in engineering endeavors. It is important that everyone involved in an engineering project understand the measurement methods and the measurement results that will be used.

Measurements can be physical, environmental, economic, operational, or some other sort of measurement that is meaningful for the particular 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 in order to generate usable data. In traditional engineering, direct measures are often used. In software engineering, a combination of both direct and derived measures is necessary [6*, p273].

The theory of measurement states that measurement is an attempt to describe an underlying

real empirical system. Measurement methods define activities that allocate a value or a 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 number (called the *measurement result)* when measuring an attribute. It follows that, to be useful, the measurement method has to be well defined. Arbitrariness in the method will reflect itself in ambiguity in the measurement results.

In some cases—particularly in the physical world—the attributes that we wish to measure are easy to grasp; however, in an artificial world like software engineering, defining the 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 or 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 vary across various time points of 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 (correct up to an 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.

3.1. Levels (Scales) of Measurement

[4*, c3s2] [6*, c7s5]

Once the operational definitions are determined, the actual measurements need to be undertaken. It is to be noted that measurement may be carried out in four different scales: namely, 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 numerals. The numerals serve only as labels, and words or letters would serve as well. The nominal scale of measurement involves only classification and the observed sampling units are put into any one of the mutually exclusive and collectively exhaustive categories (classes). Some examples of nominal scales are:

- Job titles in a company
- The software development life cycle (SDLC) model (like waterfall, iterative, agile, etc.) followed by different software projects

In nominal scale, the names of the different categories are just labels and no relationship between them is assumed. The only operations that can be carried out on nominal scale is that of counting the number of occurrences in the different classes and determining if two occurrences have the same nominal value. However, statistical analyses may be carried out to understand how entities belonging to different classes perform with respect to some other response variable.

Ordinal scale: Refers to the measurement scale where the different values obtained through the process of measurement have an implicit ordering. The intervals between values are not specified and there is no objectively defined zero element. Typical examples of measurements in ordinal scales are:

- Skill levels (low, medium, high)
- Capability Maturity Model Integration (CMMI) maturity levels of software development organizations

- Level of adherence to process as measured in a 5-point scale of excellent, above average, average, below average, and poor, indicating the range from total adherence to no adherence at all

Measurement in ordinal scale satisfies the transitivity property in the sense that if A > B and B > C, then A > C. However, arithmetic operations cannot be carried out on variables measured in ordinal scales. Thus, if we measure customer satisfaction on a 5-point ordinal scale of 5 implying a very high level of satisfaction and 1 implying a very high level of dissatisfaction, we cannot say that a score of four is twice as good as a score of two. So, it is better to use terminology such as excellent, above average, average, below average, and poor than ordinal numbers in order to avoid the error of treating an ordinal scale as a ratio scale. It is important to note that ordinal scale measures are commonly misused and such misuse can lead to erroneous conclusions [6*, p274]. A common misuse of ordinal scale measures is to present a mean and standard deviation for the data set, both of which are meaningless. However, we can find the median, as computation of the median involves counting only.

Interval scales: With the interval scale, we come to a form that is quantitative in the ordinary sense of the word. Almost all the usual statistical measures are applicable here, unless they require knowledge of a *true* zero point. The zero point on an interval scale is a matter of convention. Ratios do not make sense, but the difference between levels of attributes can be computed and is meaningful. Some examples of interval scale of measurement follow:

- Measurement of temperature in different scales, such as Celsius and Fahrenheit. Suppose T_1 and T_2 are temperatures measured in some scale. We note that the fact that T_1 is twice T_2 does not mean that one object is twice as hot as another. We also note that the zero points are arbitrary.
- Calendar dates. While the difference between dates to measure the time elapsed is a meaningful concept, the ratio does not make sense.
- Many psychological measurements aspire to create interval scales. Intelligence is often measured in interval scale, as it is not necessary to define what zero intelligence would mean.

If a variable is measured in interval scale, most of the usual statistical analyses like mean, standard deviation, correlation, and regression may be carried out on the measured values.

Ratio scale: These are quite commonly encountered in physical science. These scales of measures are characterized by the fact that operations exist for determining all 4 relations: equality, rank order, equality of intervals, and equality of ratios. Once such a scale is available, its numerical values can be transformed from one unit to another by just multiplying by a constant, e.g., conversion of inches to feet or centimeters. When measurements are being made in ratio scale, existence of a nonarbitrary zero is mandatory. All statistical measures are applicable to ratio scale; logarithm usage is valid only when these scales are used, as in the case of decibels. Some examples of ratio measures are

- the number of statements in a software program
- temperature measured in the Kelvin (K) scale or in Fahrenheit (F).

An additional measurement scale, the absolute scale, is a ratio scale with uniqueness of the measure; i.e., a measure for which no transformation is possible (for example, the number of programmers working on a project).

3.2. Direct and Derived Measures

[6*, 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 method. An example of a derived measure would be calculating the productivity of a team as the number of lines of code developed per developer-month. In both cases, the measurement method determines how to make the measurement.

3.3. Reliability and Validity

[4*, c3s4, c3s5]

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 a number of 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: namely, construct validity, criteria validity, and content validity.

3.4. Assessing Reliability

[4*, 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.

4. Engineering Design

[5*, c1s2, c1s3, c1s4]

A product's life cycle costs are largely influenced by the design of the product. This is true for manufactured products as well as for software products. The design of a software product is guided by the features to be included and the quality attributes to be provided. It is important to note that software engineers use the term "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 Design KA. The scope of engineering design is generally viewed as much broader than that of software design. The primary aim of this section is to identify the concepts needed to develop a clear understanding regarding the process of engineering 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 the 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.

4.1. 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. [8, p12]

In a similar manner, ABET defines engineering design as

> the process of devising a system, component, or process to meet desired needs. It is a decision-making process (often iterative), in which the basic sciences, mathematics, and the engineering sciences are applied to convert resources optimally to meet these stated needs. [9, p4]

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.

4.2. Design as a Problem Solving Activity

[5*, c1s4, c2s1, c3s3]

It is to be noted that engineering design is primarily a problem solving activity. Design problems are 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 [10*, c5s1].

4.3. Steps Involved in Engineering Design

[7*, c4]

Engineering problem solving begins when a need is recognized and no existing solution will meet that need. As part of this problem solving, the design goals to be achieved by the solution should be identified. Additionally, a set of acceptance criteria must be defined and used to determine how well a proposed solution will satisfy the need. Once a need for a solution to a problem has been identified, the process of engineering design has the following generic steps:

a) define the problem
b) gather pertinent information
c) generate multiple solutions
d) analyze and select a solution
e) implement the solution

All of the engineering design steps are iterative, and knowledge gained at any step in the process may be used to inform earlier tasks and trigger an iteration in the process. These steps are expanded in the subsequent sections.

a. Define the problem. At this stage, the customer's requirements are gathered. Specific information about product functions and features are also closely examined. This step includes refining the problem statement to identify the real problem to be solved and setting the design goals and criteria for success.

The problem definition is a crucial stage in engineering design. A point to note is that this step is deceptively simple. Thus, enough care must be taken to carry out this step judiciously. It is important to identify needs and link the success criteria with the required product characteristics. It is also an engineering task to limit the scope of a problem and its solution through negotiation among the stakeholders.

*b. Gather pertinent information***.** At this stage, the designer attempts to expand his/her knowledge about the problem. This is a vital, yet often neglected, stage. Gathering pertinent information can reveal facts leading to a redefinition of the

problem—in particular, mistakes and false starts may be identified. This step may also involve the decomposition of the problem into smaller, more easily solved subproblems.

While gathering pertinent information, care must be taken to identify how a product may be used as well as misused. It is also important to understand the perceived value of the product/service being offered. Included in the pertinent information is a list of constraints that must be satisfied by the solution or that may limit the set of feasible solutions.

c. Generate multiple solutions. During this stage, different solutions to the same problem are developed. It has already been stated that design problems have multiple solutions. The goal of this step is to conceptualize multiple possible solutions and refine them to a sufficient level of detail that a comparison can be done among them.

d. Analyze and select a solution. Once alternative solutions have been identified, they need to be analyzed to identify the solution that best suits the current situation. The analysis includes a functional analysis to assess whether the proposed design would meet the functional requirements. Physical solutions that involve human users often include analysis of the ergonomics or user friendliness of the proposed solution. Other aspects of the solution—such as product safety and liability, an economic or market analysis to ensure a return (profit) on the solution, performance predictions and analysis to meet quality characteristics, opportunities for incorrect data input or hardware malfunctions, and so on—may be studied. The types and amount of analysis used on a proposed solution are dependent on the type of problem and the needs that the solution must address as well as the constraints imposed on the design.

e. Implement the solution. The final phase of the design process is implementation. Implementation refers to development and testing of the proposed solution. Sometimes a preliminary, partial solution called a *prototype* may be developed initially to test the proposed design solution under certain conditions. Feedback resulting from testing a prototype may be used either to refine the design or drive the selection of an alternative design solution. One of the most important activities in design is documentation of the design solution as well as of the tradeoffs for the choices made in the design of the solution. This work should be carried out in a manner such that the solution to the design problem can be communicated clearly to others.

The testing and verification take us back to the success criteria. The engineer needs to devise tests such that the ability of the design to meet the success criteria is demonstrated. While designing the tests, the engineer must think through different possible failure modes and then design tests based on those failure modes. The engineer may choose to carry out designed experiments to assess the validity of the design.

5. Modeling, Simulation, and Prototyping

[5*, c6] [11*, c13s3] [12*, c2s3.1]

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.

5.1. 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.

5.2. 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.

5.3. 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 [6*, 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.

6. Standards

[5*, c9s3.2] [13*, c1s2]

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. [13*, p8]

Standards provide requirements, specifications, guidelines, or characteristics that must be observed 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 lets 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 with the standard must add value—real or perceived—to the product, process, or effort.

Apart from the organizational goals, standards are used for a number of 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 particular 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. Standards important to software engineers are discussed in more detail in an appendix specifically on this subject.

7. Root Cause Analysis

[4*, c5, c3s7, c9s8] [5*, c9s3, c9s4, c9s5] [13*, c13s3.4.5]

Root cause analysis (RCA) is a process designed to investigate and identify why and how an undesirable event has happened. Root causes are underlying causes. The investigator should attempt to identify specific underlying causes of the event that has occurred. The primary objective

of RCA is to prevent recurrence of the undesirable event. Thus, the more specific the investigator can be about why an event occurred, the easier it will be to prevent recurrence. A common way to identify specific underlying cause(s) is to ask a series of *why* questions.

7.1. Techniques for Conducting Root Cause Analysis

[4*, c5] [5*, c3]

There are many approaches used for both quality control and root cause analysis. The first step in any root cause analysis effort is to identify the real problem. Techniques such as statement-restatement, why-why diagrams, the revision method, present state and desired state diagrams, and the fresh-eye approach are used to identify and refine the real problem that needs to be addressed.

Once the real problem has been identified, then work can begin to determine the cause of the problem. Ishikawa is known for the seven tools for quality control that he promoted. Some of those tools are helpful in identifying the causes for a given problem. Those tools are check sheets or checklists, Pareto diagrams, histograms, run charts, scatter diagrams, control charts, and fishbone or cause-and-effect diagrams. More recently, other approaches for quality improvement and root cause analysis have emerged. Some examples of these newer methods are affinity diagrams, relations diagrams, tree diagrams, matrix charts, matrix data analysis charts, process decision program charts, and arrow diagrams. A few of these techniques are briefly described below.

A fishbone or cause-and-effect diagram is a way to visualize the various factors that affect some characteristic. The main line in the diagram represents the problem and the connecting lines represent the factors that led to or influenced the problem. Those factors are broken down into subfactors and sub-subfactors until root causes can be identified.

A very simple approach that is useful in quality control is the use of a checklist. Checklists are a list of key points in a process with tasks that must be completed. As each task is completed, it is checked off the list. If a problem occurs, then sometimes the checklist can quickly identify tasks that may have been skipped or only partially completed.

Finally, relations diagrams are a means for displaying complex relationships. They give visual support to cause-and-effect thinking. The diagram relates the specific to the general, revealing key causes and key effects.

Root cause analysis aims at preventing the recurrence of undesirable events. Reduction of variation due to common causes requires utilization of a number of techniques. An important point to note is that these techniques should be used offline and not necessarily in direct response to the occurrence of some undesirable event. Some of the techniques that may be used to reduce variation due to common causes are given below.

1. Cause-and-effect diagrams may be used to identify the sub and sub-sub causes.
2. Fault tree analysis is a technique that may be used to understand the sources of failures.
3. Designed experiments may be used to understand the impact of various causes on the occurrence of undesirable events (see Empirical Methods and Experimental Techniques in this KA).
4. Various kinds of correlation analyses may be used to understand the relationship between various causes and their impact. These techniques may be used in cases when conducting controlled experiments is difficult but data may be gathered (see Statistical Analysis in this KA).

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	Montgomery and Runger 2007 [2*]	Null and Lobur 2006 [3*]	Kan 2002 [4*]	Voland 2003 [5*]	Fairley 2009 [6*]	Tockey 2004 [7*]	McConnell 2004 [10*]	Cheney and Kincaid 2007 [11*]	Sommerville 2011 [12*]	Moore 2006 [13*]
1. Empirical Methods and Experimental Techniques	c1									
1.1. Designed Experiment										
1.2. Observational Study										
1.3. Retrospective Study										
2. Statistical Analysis	c9s1, c2s1	c10s3								
2.1. Concept of Unit of Analysis (Sampling Units), Sample, and Population	c3s6, c3s9, c4s6, c6s2, c7s1, c7s3, c8s1, c9s1									
2.2. Concepts of Correlation and Regression	c11s2, c11s8									
3. Measurement			c3s1, c3s2	c4s4	c7s5					
3.1. Levels (Scales) of Measurement			c3s2		c7s5	p442 –447				
3.2. Direct and Derived Measures										

	Montgomery and Runger 2007 [2*]	Null and Lobur 2006 [3*]	Kan 2002 [4*]	Voland 2003 [5*]	Fairley 2009 [6*]	Tockey 2004 [7*]	McConnell 2004 [10*]	Cheney and Kincaid 2007 [11*]	Sommerville 2011 [12*]	Moore 2006 [13*]
3.3. Reliability and Validity			c3s4, c3s5							
3.4. Assessing Reliability			c3s5							
4. Engineering Design				c1s2, c1s3, c1s4						
4.1. Design in Engineering Education										
4.2. Design as a Problem Solving Activity				c1s4, c2s1, c3s3			c5s1			
4.3. Steps Involved in Engineering Design						c4				
5. Modeling, Prototyping, and Simulation				c6				c13s3	c2 s3.1	
5.1. Modeling										
5.2. Simulation										
5.3. Prototyping										
6. Standards				c9 s3.2						c1s2
7. Root Cause Analysis			c5, c3s7, c9s8	c9s3, c9s4, c9s5						c13 s3.4.5
7.1. Techniques for Conducting Root Cause Analysis			c5	c3						

FURTHER READINGS

A. Abran, *Software Metrics and Software Metrology.* [14]

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.* [15]

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] *ISO/IEC/IEEE 24765:2010 Systems and Software Engineering—Vocabulary*, ISO/IEC/IEEE, 2010.

[2*] D.C. Montgomery and G.C. Runger, *Applied Statistics and Probability for Engineers*, 4th ed., Wiley, 2007.

[3*] L. Null and J. Lobur, *The Essentials of Computer Organization and Architecture*, 2nd ed., Jones and Bartlett Publishers, 2006.

[4*] S.H. Kan, *Metrics and Models in Software Quality Engineering*, 2nd ed., Addison-Wesley, 2002.

[5*] G. Voland, *Engineering by Design*, 2nd ed., Prentice Hall, 2003.

[6*] R.E. Fairley, *Managing and Leading Software Projects*, Wiley-IEEE Computer Society Press, 2009.

[7*] S. Tockey, *Return on Software: Maximizing the Return on Your Software Investment*, Addison-Wesley, 2004.

[8] Canadian Engineering Accreditation Board, Engineers Canada, "Accreditation Criteria and Procedures," Canadian Council of Professional Engineers, 2011; www.engineerscanada.ca/files/w_Accreditation_Criteria_Procedures_2011.pdf.

[9] ABET Engineering Accreditation Commission, "Criteria for Accrediting Engineering Programs, 2012-2013," ABET, 2011; www.abet.org/uploadedFiles/Accreditation/Accreditation_Process/Accreditation_Documents/Current/eac-criteria-2012-2013.pdf.

[10*] S. McConnell, *Code Complete*, 2nd ed., Microsoft Press, 2004.

[11*] E.W. Cheney and D.R. Kincaid, *Numerical Mathematics and Computing*, 6th ed., Brooks/Cole, 2007.

[12*] I. Sommerville, *Software Engineering*, 9th ed., Addison-Wesley, 2011.

[13*] J.W. Moore, *The Road Map to Software Engineering: A Standards-Based Guide*, Wiley-IEEE Computer Society Press, 2006.

[14] A. Abran, *Software Metrics and Software Metrology*, Wiley-IEEE Computer Society Press, 2010.

[15] W.G. Vincenti, *What Engineers Know and How They Know It*, John Hopkins University Press, 1990.

APPENDIX A

KNOWLEDGE AREA DESCRIPTION SPECIFICATIONS

INTRODUCTION

This document presents the specifications provided to the Knowledge Area Editors (KA Editors) regarding the Knowledge Area Descriptions (KA Descriptions) of the Version 3 (V3) edition of the *Guide to the Software Engineering Body of Knowledge (SWEBOK Guide)*. This document will also enable readers, reviewers, and users to clearly understand what specifications were used when developing this version of the *SWEBOK Guide*.

This document begins by situating the *SWEBOK Guide* as a foundational document for the IEEE Computer Society suite of software engineering products and more widely within the software engineering community at large. The role of the baseline and the Change Control Board is then described. 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. Noncontent issues such as submission format and style guidelines are also 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 suite of software engineering products. The *SWEBOK Guide* is also more widely recognized as a foundational document within the software engineering community at large notably through the official recognition of the 2004 Version as ISO/IEC Technical Report 19759:2005. The list of knowledge areas (KAs) and the breakdown of topics within each KA is described and detailed in the introduction of this *SWEBOK Guide*.

Consequently, the *SWEBOK Guide* is foundational to other initiatives within the IEEE Computer Society:

a) The list of KAs and the breakdown of topics within each KA are also adopted by the software engineering certification and associated professional development products offered by the IEEE Computer Society (see www.computer.org/certification).

b) The list of KAs and the breakdown of topics are also foundational to the software engineering curricula guidelines developed or endorsed by the IEEE Computer Society (www.computer.org/portal/web/education/Curricula).

c) The Consolidated Reference List (see Appendix C), meaning the list of recommended reference materials (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 BOARD

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 comprised of the list of KAs, the breakdown of

topics within each KA, and the Consolidated Reference List.

A Change Control Board (CCB) has been in place for the development of this version to handle all 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 CCB before being implemented. This CCB is comprised of members of the initiatives listed above and acting under the authority of the Software and Systems Engineering Committee of the IEEE Computer Society Professional Activities Board.

CRITERIA AND REQUIREMENTS FOR THE BREAKDOWN OF TOPICS WITHIN A KNOWLEDGE AREA

a) KA Editors are instructed to adopt the baseline breakdown of topics.
b) The breakdown of topics is expected to be "reasonable," not "perfect."
c) The breakdown of topics within a KA must decompose the subset of the Software Engineering Body of Knowledge that is "generally recognized." See below for a more detailed discussion of this point.
d) 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.
e) The breakdown of topics must, as much as possible, be compatible with the various schools of thought within software engineering.
f) 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.
g) The breakdown of topics is expected to be as inclusive as possible.
h) The *SWEBOK Guide* adopts the position that even though the following "themes" are common across all Knowledge Areas, they are also an integral part of all Knowledge Areas and therefore must be incorporated into the proposed breakdown of topics of each Knowledge Area. These common themes are measurement, quality (in general), and security.
i) 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.
j) Topic names must be significant enough to be meaningful even when cited outside the *SWEBOK Guide*.
k) The description of a KA will include a chart (in tree form) describing the knowledge breakdown.

CRITERIA AND REQUIREMENTS FOR DESCRIBING TOPICS

Topics need only be sufficiently described so the reader can select the appropriate reference material according to his/her needs. Topic descriptions must not be prescriptive.

CRITERIA AND REQUIREMENTS FOR REFERENCE MATERIAL

a) 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.
b) There are three categories of reference material:

- Recommended References. The set of Recommended References (to the level of section number) is collectively known as the Consolidated Reference List.
- Further Readings.
- Additional references cited in the KA Description (for example, the source of a quotation or reference material in support of a rationale behind a particular argument).

c) 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.

d) 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.

e) Reference material must be generally available and must not be confidential in nature.

f) Reference material must be in English.

g) 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 nor conflicting practices
iv. credible: recognized as providing expert treatment
v. current: treating the subject in a manner that is commensurate with currently generally accepted knowledge
vi. succinct: as short as possible (both in 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. Exceptionally, a topic may be self-descriptive and not cite a reference material item (for example, 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 by identifying what specific chapter or section is relevant.

» A matrix of reference material (to the level of section number) versus topics must be provided.

» 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 material were written in a coherent manner that followed the proposed breakdown of topics and in a uniform style (for example, in a new book based on the proposed KA description), an average target across all KAs for the number of pages would be 750. However, this target may not be attainable when selecting existing reference material due to differences in style and overlap and redundancy between the selected reference materials.
ii. In other words, the target for the number of pages for the entire collection of recommended references of 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 on this KA for a software engineering licensing exam that a graduate would pass after completing four years of work experience.

h) Additional reference material can be included by the KA Editor in a "Further Readings" list:

» These further readings must be related to the topics in the breakdown rather than, for example, to more advanced topics.
» The list must be annotated (within 1 paragraph per reference) as to why this reference material was included in the list of further readings. Further readings could include: new versions of an existing reference already included in the recommended references, 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 readings and the breakdown of topics.

i) 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 while maintaining its technical excellence. Additional reference material should therefore only be brought in by the KA Editor if it is necessary to the discussion. Examples are to identify the source of a quotation or to cite reference item in support of a rationale behind a particular and important argument.

COMMON STRUCTURE

KA descriptions should use the following structure:

- Acronyms
- Introduction
- Breakdown of Topics of the KA (including a figure describing the breakdown)
- Matrix of Topics vs. Reference Material
- List of Further Readings
- 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 being:

> 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 USA) 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 and but what they expect will be "generally recognized" in a 3- to 5-year timeframe.

Specialized Practices Used Only for Certain Types of Software	**Generally Recognized** Established traditional practices recommended by many organizations
	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, appendices, 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; www.gswe2009.org. [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. *IEEE Std. 12207-2008 (a.k.a. ISO/IEC 12207:2008) Standard for Systems and Software Engineering—Software Life Cycle Processes*, IEEE, 2008 [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 as a pivotal standard by the Software and System Engineering Standards Committee (S2ESC) of the IEEE.

Even though we do not intend that the *Guide to the Software Engineering Body of Knowledge* 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. J.W. Moore, *The Road Map to Software Engineering: A Standards-Based Guide*, Wiley-IEEE Computer Society Press, 2006. [4*]

This book describes the scope, roles, uses, and development trends of the most widely used software engineering standards. It concentrates on important software engineering activities—quality and project management, system engineering, dependability, and safety. The analysis and regrouping of the standard collections exposes the reader to key relationships between standards.

Even though the *SWEBOK Guide* is not a software engineering standard per se, special care will be taken throughout the document regarding the compatibility of the *Guide* with the current IEEE and ISO/IEC Systems and Software Engineering Standards Collection.

4. *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*, IEEE Computer Society and Association for Computing Machinery, 2004; http://sites.computer.org/ccse/SE2004Volume.pdf. [5]

This document describes curriculum guidelines for an undergraduate degree in software engineering. The *SWEBOK Guide* is identified as being "one of the primary sources" in developing the body of knowledge underlying these guidelines.

5. *ISO/IEC/IEEE 24765:2010 Systems and Software Engineering—Vocabulary*, ISO/IEC/IEEE, 2010; www.computer.org/sevocab. [6]

The hierarchy of references for terminology is *Merriam Webster's Collegiate Dictionary* (11th ed.) [7], IEEE/ISO/IEC 24765 [6], and new proposed definitions if required.

6. "Certification and Training for Software Professionals," IEEE Computer Society, 2013; www.computer.org/certification. [8]

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.

STYLE AND TECHNICAL GUIDELINES

- KA Descriptions should conform to the Word template available at www.computer.org/portal/web/cscps/formatting.
- KA Descriptions are expected to follow the IEEE Computer Society Style Guide (www.computer.org/portal/web/publications/styleguide).
- Files are to be submitted in Microsoft Word format.
- All citations of reference material are to be produced using EndNote Web as indicated in the instructions provided to KA Editors in this regard.

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 would ensure 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 due to 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 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(R) Guide)*, 5th ed., Project Management Institute, 2013.

[2] Integrated Software and Systems Engineering Curriculum (iSSEc) Project, *Graduate Software Engineering 2009 (GSwE2009): Curriculum Guidelines for Graduate Degree Programs in Software Engineering*, Stevens Institute of Technology, 2009; www.gswe2009.org.

[3] *IEEE Std. 12207-2008 (a.k.a. ISO/IEC 12207:2008) Standard for Systems and Software Engineering—Software Life Cycle Processes, IEEE, 2008.*

[4*] J.W. Moore, *The Road Map to Software Engineering: A Standards-Based Guide*, Wiley-IEEE Computer Society Press, 2006.

[5] Joint Task Force on Computing Curricula, IEEE Computer Society and Association for Computing Machinery, *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*, 2004; http://sites.computer.org/ccse/SE2004Volume.pdf.

[6] *ISO/IEC/IEEE 24765:2010 Systems and Software Engineering—Vocabulary*, ISO/IEC/IEEE, 2010.

[7] *Merriam-Webster's Collegiate Dictionary*, 11th ed., 2003.

[8] IEEE Computer Society, "Certification and Training for Software Professionals," 2013; www.computer.org/certification.

APPENDIX B

IEEE AND ISO/IEC STANDARDS SUPPORTING THE SOFTWARE ENGINEERING BODY OF KNOWLEDGE (SWEBOK)

Some might say that the supply of software engineering standards far exceeds the demand. One seldom listens to a briefing on the subject without suffering some apparently obligatory joke that there are too many of them. However, the existence of standards takes a very large (possibly infinite) trade space of alternatives and reduces that space to a smaller set of choices—a huge advantage for users. Nevertheless, it can still be difficult to choose from dozens of alternatives, so supplementary guidance, like this appendix, can be helpful. A summary list of the standards mentioned in this appendix appears at the end.

To reduce tedium in reading, a few simplifications and abridgements are made in this appendix:

- ISO/IEC JTC 1/SC 7 maintains nearly two hundred standards on the subject. IEEE maintains about fifty. The two organizations are in the tenth year of a systematic program to coordinate and integrate their collections. In general, this article will focus on the standards that are recognized by both organizations, taking this condition as evidence that wide agreement has been obtained. Other standards will be mentioned briefly.
- Standards tend to have long, taxonomical titles. If there were a single standard for building an automobile, the one for your Camry probably would be titled something like, "Vehicle, internal combustion, four-wheel, passenger, sedan." Also, modern standards organizations provide their standards from databases. Like any database, these sometimes contain errors, particularly for the titles. So this article will often paraphrase the title of the standard or simply use its number. In obtaining a standard of interest, the reader should rely on the number, not the title, given in this article. For reasons of consistency, the article will use the IEEE's convention for the capitalization of titles—nouns, pronouns, adjectives, verbs, adverbs, and first and last words have an initial capital letter—despite the fact that IEEE and ISO/IEC use differing conventions.
- Because these standards are being continually revised to take account of new technologies and usage patterns, this article will be obsolescent before it is published. Therefore, it will occasionally discuss standards that have not yet been published, if they are likely to assume significant importance.
- Explicit trademarks are omitted. Suffice it to say that IEEE places a trademark on all of its standards' designations.

There are some other conventions of interest:

- In both IEEE and ISO/IEC, standards for *systems* engineering are maintained by the same committee as those for *software* engineering. Many of the standards apply to both. So, instead of making fine distinctions, this article will deal with both.
- On the other hand, both S2ESC and SC 7 (see below for descriptions of these organizations) are responsible for standards that don't qualify as "engineering." In the US and many other countries, the services of a licensed engineer are required when a product might affect public safety, health,

and welfare as opposed to affecting merely the pocketbook of the client. This appendix will respect that distinction and ignore standards that appear to be merely economic in consequence.

- User documentation is assumed to be developed similarly to software. For example, a standard concerning the design of user documentation is described in the Software Design KA.
- Some jointly developed standards are explicitly labeled as joint developments, e.g., ISO/IEC/IEEE 24765. In other cases, the standards have different designations in the two organizations. Examples include

  » IEEE Std. 12207:2008 (a.k.a. ISO/IEC 12207:2008), where "a.k.a." ("also known as") is this appendix's abbreviation to note the designation in the other organization;
  » IEEE Std. 15939:2008 Standard Adoption of ISO/IEC 15939:2007, an adoption by IEEE of a standard developed in ISO/IEC;
  » IEEE Std. 1220:2005 (a.k.a. ISO/IEC 26702:2007), a "fast-track" by ISO/IEC of a standard developed in IEEE.

  In each of these cases, the standards are substantively identical in the two organizations, differing only in front matter and, occasionally, added informational material.

A summary list of all of the mentioned standards is provided at the end of this appendix.

ISO/IEC JTC 1/SC 7, SOFTWARE AND SYSTEMS ENGINEERING

ISO/IEC JTC 1/SC 7 is the major source of international standards on software and systems engineering. Its name is formed taxonomically. Joint Technical Committee 1 (JTC 1) is a child of the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC); it has the scope of "information technology" and subdivides its work among a number of subcommittees; Subcommittee 7 (SC 7) is the one responsible for software and systems engineering. SC 7, and its working groups, meets twice a year, attracting delegations representing the national standards bodies of participating nations. Each nation follows its own procedures for determining national positions and each nation has the responsibility of determining whether an ISO/IEC standard should be adopted as a national standard.

SC 7 creates three types of documents:

- International Standards: Documents containing requirements that must be satisfied in order to claim conformance.
- Technical Specifications (formerly called Technical Reports, type 1 and type 2): Documents published in a preliminary manner while work continues.
- Technical Reports (formerly called Technical Reports, type 3): Documents inherently unsuited to be standards, usually because they are descriptive rather than prescriptive.

The key thing to remember is that only the first category counts as a consensus standard. The reader can easily recognize the others by the suffix TS or TR prepended to the number of the document.

IEEE SOFTWARE AND SYSTEMS ENGINEERING STANDARDS COMMITTEE (S2ESC)

IEEE is the world's largest organization of technical professionals, with about 400,000 members in more than 160 countries. The publication of standards is performed by the IEEE Standards Association (IEEE-SA), but the committees that draft and sponsor the standards are in the various IEEE societies; S2ESC is a part of the IEEE Computer Society. IEEE is a global standards maker because its standards are used in many different countries. Despite its international membership (about 50% non-US), though, the IEEE-SA routinely submits its standards to the American National Standards Institute (ANSI) for endorsement as "American National Standards." Some S2ESC standards are developed within S2ESC, some are developed jointly with SC 7, and some are adopted after being developed by SC 7.

IEEE-SA publishes three types of "standards":

- Standards, with a preponderance of the verb "shall"
- Recommended Practices, with a preponderance of the verb "should"
- Guides, with a preponderance of the verb "may."

All three of these compare to ISO/IEC standards. IEEE-SA does have the concept of a "Trial-Use" standard, which is roughly comparable to an ISO/IEC Technical Specification. However, it has nothing comparable to an ISO/IEC Technical Report; one would look elsewhere in IEEE for documents of this ilk.

THE STANDARDS

The remainder of this article allocates the selected standards to relevant knowledge areas (KAs) of the *SWEBOK Guide*. There is a section for each KA. Within each section, the relevant standards are listed—the ones that principally apply to the KA as well as others that principally apply to other KAs but which are also related to the current one. Following each standard is a brief summary. In most cases, the summary is a quotation or paraphrase of the abstract or other introductory material from the text of the standard.

Most of the standards easily fit into one KA. Some fit into more than one; in such cases, a cross-reference is provided. Two standards apply to all KAs, so they are listed in a category called "General." All of the standards related to computer-aided software engineering (CASE) tools and environments are listed in the Software Engineering Models and Methods KA section.

GENERAL

The first two standards are so central that they could be slotted into all of the KAs. Two more are described in the Software Engineering Process KA, but are mentioned here because they provide a helpful framework and because the descriptions of several other standards refer to them.

ISO/IEC TR 19759 is the *SWEBOK Guide* itself. It's not an IEEE standard because, lacking prescriptive verbs, it doesn't satisfy the criteria for any of the IEEE categories. In ISO/IEC, it is a "technical report"—defined as a document inherently unsuited to be a standard. The 2004 IEEE *SWEBOK Guide* was adopted by ISO/IEC without change. Presumably, ISO/IEC will adopt Version 3 of the *SWEBOK Guide*.

ISO/IEC TR 19759:2005 Software Engineering—*Guide to the Software Engineering Body of Knowledge* (SWEBOK)

Applies to all KAs

ISO/IEC 19759:2005, a *Guide to the Software Engineering Body of Knowledge (SWEBOK)*, identifies and describes that subset of the body of knowledge that is generally accepted, even though software engineers must be knowledgeable not only in software engineering, but also, of course, in other related disciplines. SWEBOK is an all-inclusive term that describes the sum of knowledge within the profession of software engineering.

The text of the *SWEBOK Guide* is freely available at www.swebok.org/. The ISO/IEC adoption of the *Guide* is freely available at http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html.

ISO/IEC/IEEE 24765 provides a shared vocabulary for the systems and software engineering standards of both SC 7 and S2ESC.

ISO/IEC/IEEE 24765:2010 Systems and Software Engineering—Vocabulary

Applies to all KAs

ISO/IEC/IEEE 24765:2010 provides a common vocabulary applicable to all systems and software engineering work. It was prepared to collect and support the standardization of terminology. ISO/IEC/IEEE 24765:2010 is intended to serve as a useful reference for those in the information technology field and to encourage the use of systems and software engineering standards prepared by ISO and liaison organizations IEEE Computer Society and Project Management Institute. ISO/IEC/IEEE 24765:2010 includes references to the

active source standards for each definition so that the use of the term can be further explored.

The vocabulary is descriptive, rather than prescriptive; it gathers up all of the definitions from all of the relevant standards, as well as a few other sources, rather than choosing among competing definitions.

The content of the 24765 standard is freely accessible online at www.computer.org/sevocab.

Two standards, 12207 and 15288, provide a complete set of processes for the entire life cycle of a system or a software product. The two standards are aligned for concurrent use on a single project or in a single organization. They are mentioned here because they are often used as a framework for explaining or localizing the role of other standards in the life cycle.

IEEE Std. 12207-2008 (a.k.a. ISO/IEC 12207:2008) Standard for Systems and Software Engineering—Software Life Cycle Processes

See Software Engineering Process KA

IEEE Std. 15288-2008 (a.k.a. ISO/IEC 15288:2008) Standard for Systems and Software Engineering—System Life Cycle Processes

See Software Engineering Process KA

SOFTWARE REQUIREMENTS

The primary standard for software and systems requirements engineering is a new one that replaced several existing IEEE standards. It provides a broad view of requirements engineering across the entire life cycle.

ISO/IEC/IEEE 29148:2011 Systems and Software Engineering—Life Cycle Processes—Requirements Engineering

ISO/IEC/IEEE 29148:2011 contains provisions for the processes and products related to the engineering of requirements for systems and software products and services throughout the life cycle. It defines the construct of a good requirement, provides attributes and characteristics of requirements, and discusses the iterative and recursive application of requirements processes throughout the life cycle. ISO/IEC/IEEE 29148:2011 provides additional guidance in the application of requirements engineering and management processes for requirements-related activities in ISO/IEC 12207:2008 and ISO/IEC 15288:2008. Information items applicable to the engineering of requirements and their content are defined. The content of ISO/IEC/IEEE 29148:2011 can be added to the existing set of requirements-related life cycle processes defined by ISO/IEC 12207:2008 or ISO/IEC 15288:2008, or it can be used independently.

A multipart ISO/IEC standard provides principles and methods for "sizing" software based on its requirements. The functional size is often useful in the denominator of measurements of quality and productivity in software development. It may also play a role in contracting for service-level agreements.

ISO/IEC 14143 [six parts] Information Technology—Software Measurement—Functional Size Measurement

ISO/IEC 14143 describes FSM (functional size measurement). The concepts of functional size measurement (FSM) are designed to overcome the limitations of earlier methods of sizing software by shifting the focus away from measuring how the software is implemented to measuring size in terms of the functions required by the user.

FSM is often known as "function point counting." The four standards listed below are alternative methods for function point counting—all meet the requirements of ISO/IEC 14143. The dominant method, in terms of market share, is the IFPUG method, described in ISO/IEC 20926. Other methods are variations intended to improve the validity of the count in various circumstances. For example, ISO/IEC 19761—COSMIC is

notably intended to be used on software with a real-time component.

ISO/IEC 19761:2011 Software Engineering—COSMIC: A Functional Size Measurement Method

ISO/IEC 20926:2009 Software and Systems Engineering—Software Measurement—IFPUG Functional Size Measurement Method

ISO/IEC 20968:2002 Software Engineering—Mk II Function Point Analysis—Counting Practices Manual

ISO/IEC 24570:2005 Software Engineering—NESMA Functional Size Measurement Method Version 2.1—Definitions and Counting Guidelines for the Application of Function Point Analysis

Sometimes requirements are described in natural language, but sometimes they are described in formal or semiformal notations. The objective of the Unified Modeling Language (UML) is to provide system architects, software engineers, and software developers with tools for analysis, design, and implementation of software-based systems as well as for modeling business and similar processes. The two parts of ISO/IEC 19505 define UML, revision 2. The older ISO/IEC 19501 is an earlier version of UML. They are mentioned here because they are often used to model requirements.

ISO/IEC 19501:2005 Information Technology—Open Distributed Processing—Unified Modeling Language (UML) Version 1.4.2

See Software Engineering Models and Methods KA

ISO/IEC 19505:2012 [two parts] Information Technology—Object Management Group Unified Modeling Language (OMG UML)

See Software Engineering Models and Methods KA

SOFTWARE DESIGN

The software design KA includes both software architectural design (for determining the relationships among the items of the software and detailed design (for describing the individual items). ISO/IEC/IEEE 42010 concerns the description of architecture for systems and software.

ISO/IEC/IEEE 42010:2011 Systems and Software Engineering—Architecture Description

ISO/IEC/IEEE 42010:2011 addresses the creation, analysis, and sustainment of architectures of systems through the use of architecture descriptions. A conceptual model of architecture description is established. The required contents of an architecture description are specified. Architecture viewpoints, architecture frameworks and architecture description languages are introduced for codifying conventions and common practices of architecture description. The required content of architecture viewpoints, architecture frameworks and architecture description languages is specified. Annexes provide the motivation and background for key concepts and terminology and examples of applying ISO/IEC/IEEE 42010:2011.

Like ISO/IEC/IEEE 42010, the next standard treats software "design" as an abstraction, independent of its representation in a document. Accordingly, the standard places provisions on the description of design, rather than on design itself.

IEEE Std. 1016-2009 Standard for Information Technology—Systems Design—Software Design Descriptions

This standard describes software designs and establishes the information content and organization of a software design description (SDD). An SDD is a representation of a software design to be used for recording design information and communicating that design information to key design

stakeholders. This standard is intended for use in design situations in which an explicit software design description is to be prepared. These situations include traditional software construction activities (when design leads to code) and reverse engineering situations (when a design description is recovered from an existing implementation). This standard can be applied to commercial, scientific, or military software that runs on digital computers. Applicability is not restricted by the size, complexity, or criticality of the software. This standard can be applied to the description of high-level and detailed designs. This standard does not prescribe specific methodologies for design, configuration management, or quality assurance. This standard does not require the use of any particular design languages, but establishes requirements on the selection of design languages for use in an SDD. This standard can be applied to the preparation of SDDs captured as paper documents, automated databases, software development tools, or other media.

By convention, this appendix treats user documentation as a part of a software system. Therefore, the various aspects of user documentation—its design, its testing, and so forth—are allocated to different KAs. The next standard deals with the design of user documentation.

IEEE Std. 26514-2010 Standard Adoption of ISO/IEC 26514:2008 Systems and Software Engineering—Requirements for Designers and Developers of User Documentation

This standard provides requirements for the design and development of software user documentation as part of the life cycle processes. It defines the documentation process from the viewpoint of the documentation developer and also covers the documentation product. It specifies the structure, content, and format for user documentation and also provides informative guidance for user documentation style. It is independent of the software tools that may be used to produce documentation and applies to both printed documentation and onscreen documentation. Much of this standard is also applicable to user documentation for systems including hardware.

SOFTWARE CONSTRUCTION

The term "software construction" refers to the detailed creation of working, meaningful software through a combination of coding, verification, unit testing, integration testing, and debugging.

There are few standards on the details of software coding. It has been found through (mostly bad) experience that coding conventions are not appropriate for standardization because, in most cases, the real benefit comes from the consistency of applying an arbitrary convention rather than the convention itself. So, although coding conventions are a good idea, it is generally left to the organization or the project to develop such a standard.

Nevertheless, the subject of secure coding has attracted attention in recent years because some coding idioms are insecure in the face of attack. A Technical Report prepared by ISO/IEC JTC 1/SC 22 (programming languages) describes vulnerabilities in programming languages and how they can be avoided.

ISO/IEC TR 24772:2013 Information Technology—Programming Languages—Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use

ISO/IEC TR 24772:2013 specifies software programming language vulnerabilities to be avoided in the development of systems where assured behavior is required for security, safety, mission-critical, and business-critical software. In general, this guidance is applicable to the software developed, reviewed, or maintained for any application.

Vulnerabilities are described in a generic manner that is applicable to a broad range of programming languages. Annexes relate the generic guidance to a selection of specific programming languages.

The Technical Report is freely available at http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html.

Two standards are mentioned here because unit testing is often regarded as an activity of software construction. IEEE and ISO/IEC are cooperating in the development of a four-part joint standard, 29119, that will provide a comprehensive treatment of testing and supplant IEEE Std. 1008.

IEEE Std. 1008-1987 Standard for Software Unit Testing

See Software Testing KA

ISO/IEC/IEEE 29119 [four parts] (Draft) Software and Systems Engineering—Software Testing

See Software Testing KA

The next standard provides for the development of user documentation during an agile development process. It is mentioned here because agile development is sometimes regarded as construction.

ISO/IEC/IEEE 26515:2012 Systems and Software Engineering—Developing User Documentation in an Agile Environment

See Software Engineering Models and Methods KA

Coding is not the only way to create a software product. Often code (as well as requirements and design) is reused from previous projects or engineered for reuse in future projects. IEEE Std. 1517 is mentioned here because it provides a common framework for extending the system and software life cycle processes of IEEE Std. 12207:2008 to include the systematic practice of reuse.

IEEE Std. 1517-2010 Standard for Information Technology—System and Software Life Cycle Processes—Reuse Processes

See Software Engineering Process KA

SOFTWARE TESTING

Oddly, there are few standards for testing. IEEE Std. 829 is the most comprehensive.

IEEE Std. 829-2008 Standard for Software and System Test Documentation

Test processes determine whether the development products of a given activity conform to the requirements of that activity and whether the system and/or software satisfies its intended use and user needs. Testing process tasks are specified for different integrity levels. These process tasks determine the appropriate breadth and depth of test documentation. The documentation elements for each type of test documentation can then be selected. The scope of testing encompasses software-based systems, computer software, hardware, and their interfaces. This standard applies to software-based systems being developed, maintained, or reused (legacy, commercial off-the-shelf, nondevelopmental items). The term "software" also includes firmware, microcode, and documentation. Test processes can include inspection, analysis, demonstration, verification, and validation of software and software-based system products.

IEEE Std. 1008 focuses on unit testing.

IEEE Std. 1008-1987 Standard for Software Unit Testing

The primary objective is to specify a standard approach to software unit testing that can be used as a basis for sound software engineering practice. A second objective is to describe the software engineering concepts and testing assumptions on which the standard approach is based. A third objective is to provide guidance and resource information to assist with the implementation and usage of the standard unit testing approach.

IEEE and ISO/IEC JTC 1/SC 7 are cooperating in a project to develop a single comprehensive standard that covers all aspects of testing. One can hope for publication of the four-part standard by 2014. Portions of the content remain controversial. One taxonomical issue is whether "static methods"—such as inspection, review, and static analysis—should fall within the scope of "testing" or should be distinguished as "verification and validation." Although the resolution of the issue is probably of little importance to users of the standard, it assumes great importance to the standards-writers who must manage an integrated suite of interoperating standards.

ISO/IEC/IEEE 29119 [four parts] (Draft) Software and Systems Engineering—Software Testing

The purpose of ISO/IEC 29119 Software Testing is to define an internationally agreed standard for software testing that can be used by any organization when performing any form of software testing.

Testing of user documentation is described in the next standard, providing requirements for the test and review of software user documentation as part of the life cycle processes. It defines the documentation process from the viewpoint of the documentation tester and reviewer. It is relevant to roles involved in testing and development of software and user documentation, including project managers, usability experts, and information developers in addition to testers and reviewers.

IEEE Std. 26513-2010 Standard Adoption of ISO/IEC 26513:2009 Systems and Software Engineering—Requirements for Testers and Reviewers of Documentation

ISO/IEC 26513 provides the minimum requirements for the testing and reviewing of user documentation, including both printed and onscreen documents used in the work environment by the users of systems software. It applies to printed user manuals, online help, tutorials, and user reference documentation.

It specifies processes for use in testing and reviewing of user documentation. It is not limited to the test and review phase of the life cycle, but includes activities throughout the information management and documentation management processes.

Two standards are mentioned here because some sources consider software verification and validation to be taxonomically included in testing.

IEEE Std. 1012-2012 Standard for System and Software Verification and Validation

See Software Quality KA

IEEE Std. 1044-2009 Standard for Classification for Software Anomalies

See Software Quality KA

SOFTWARE MAINTENANCE

This standard—the result of harmonizing distinct IEEE and ISO/IEC standards on the subject—describes a single comprehensive process for the management and execution of software maintenance. It expands on the provisions of the software maintenance process provided in ISO/IEC/IEEE 12207.

IEEE Std. 14764-2006 (a.k.a. ISO/IEC 14764:2006) Standard for Software Engineering—Software Life Cycle Processes—Maintenance

ISO/IEC 14764:2006 describes in greater detail management of the maintenance process described in ISO/IEC 12207, including amendments. It also establishes definitions for the various types of maintenance. ISO/IEC 14764:2006 provides guidance that applies to planning, execution and control, review and evaluation, and closure of the maintenance process. The scope of ISO/IEC 14764:2006 includes maintenance for multiple software products with the same maintenance resources. "Maintenance" in ISO/IEC 14764:2006 means software maintenance unless otherwise stated.

ISO/IEC 14764:2006 provides the framework within which generic and specific software maintenance plans may be executed, evaluated, and tailored to the maintenance scope and magnitude of given software products. It provides the framework, precise terminology, and processes to allow the consistent application of technology (tools, techniques, and methods) to software maintenance.

It does not address the operation of software and the operational functions, e.g., backup, recovery, and system administration, which are normally performed by those who operate the software.

ISO/IEC 14764:2006 is written primarily for maintainers of software and additionally for those responsible for development and quality assurance. It may also be used by acquirers and users of systems containing software, who may provide inputs to the maintenance plan.

SOFTWARE CONFIGURATION MANAGEMENT

There is one standard for configuration management.

IEEE Std. 828-2012 Standard for Configuration Management in Systems and Software Engineering

This standard establishes the minimum requirements for processes for configuration management (CM) in systems and software engineering. The application of this standard applies to any form, class, or type of software or system. This revision of the standard expands the previous version to explain CM, including identifying and acquiring configuration items, controlling changes, reporting the status of configuration items, as well as software builds and release engineering. Its predecessor defined only the contents of a software configuration management plan. This standard addresses what CM activities are to be done, when they are to happen in the life cycle, and what planning and resources are required. It also describes the content areas for a CM plan. The standard supports ISO/IEC/IEEE 12207:2008 and ISO/IEC/IEEE 15288:2008 and adheres to the terminology in ISO/IEC/IEEE Std. 24765 and the information item requirements of IEEE Std. 15939.

ISO/IEC JTC 1/SC 7 has not yet determined what action it should take regarding the new IEEE Std. 828. There are issues concerning the extent of compatibility with ISO/IEC/IEEE 12207 and other standards in the SC 7 suite. It should be noted, though, that SC 7 does not have a competing standard.

SOFTWARE ENGINEERING MANAGEMENT

Most readers will interpret the phrase "software engineering management" to mean the management of a *project* that concerns software. There are at least two possible extensions to this generalization, though. Some software activities are managed according to a service-level agreement (SLA). SLAs do not meet the criteria for "project" according to some definitions. Also, it has become generally agreed that some management of software should occur in the organization at a level above the project, so that all projects can benefit from a common investment. A commonly cited example is the provision of software processes and tooling by the organization.

Software project management can be regarded as a specialization of "project management"—often regarded as a distinct discipline. The Project Management Institute's *Guide to the Project Management Body of Knowledge (PMBOK® Guide)* is often regarded as the authoritative source for this knowledge. From time to time, IEEE adopts the most recent version of the *PMBOK® Guide* as an IEEE standard.

IEEE Std. 1490-2011 Guide—Adoption of the Project Management Institute (PMI®) Standard, A Guide to the Project Management Body of Knowledge (PMBOK® Guide)—Fourth Edition

The *PMBOK® Guide* identifies 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 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. The *PMBOK® Guide* also provides and promotes a common vocabulary within the project management profession for discussing, writing, and applying project management concepts. Such a standard vocabulary is an essential element of a professional discipline. The Project Management Institute (PMI) views this standard as a foundational project management reference for its professional development programs and certifications.

The 2008 revisions of ISO/IEC/IEEE 12207 and 15288 provide project management processes for software and systems and relate them to organization-level processes as well as technical processes. The jointly developed 16326 standard, replacing two older standards, expands those provisions with guidance for application.

ISO/IEC/IEEE 16326:2009 Systems and Software Engineering—Life Cycle Processes—Project Management

ISO/IEC/IEEE 16326:2009 provides normative content specifications for project management plans covering software projects and software-intensive system projects. It also provides detailed discussion and advice on applying a set of project processes that are common to both the software and system life cycle as covered by ISO/IEC 12207:2008 (IEEE Std. 12207-2008) and ISO/IEC 15288:2008 (IEEE Std. 15288-2008), respectively. The discussion and advice are intended to aid in the preparation of the normative content of project management plans. ISO/IEC/IEEE 16326:2009 is the result of the harmonization of ISO/IEC TR 16326:1999 and IEEE Std. 1058-1998.

Particularly in high-technology applications and high-consequence projects, the management of risk is an important aspect of the overall project management responsibilities. This standard deals with that subject.

IEEE Std. 16085-2006 (a.k.a. ISO/IEC 16085:2006) Standard for Systems and Software Engineering—Software Life Cycle Processes—Risk Management

ISO/IEC 16085:2006 defines a process for the management of risk in the life cycle. It can be added to the existing set of system and software life cycle processes defined by ISO/IEC 15288 and ISO/IEC 12207, or it can be used independently.

ISO/IEC 16085:2006 can be applied equally to systems and software.

The purpose of risk management is to identify potential managerial and technical problems before they occur so that actions can be taken that reduce or eliminate the probability and/or impact of these problems should they occur. It is a critical tool for continuously determining the feasibility of project plans, for improving the search for and identification of potential problems that can affect life cycle activities and the quality and performance of products, and for improving the active management of projects.

The analysis of risk and risk mitigation depends crucially upon measurement. This international standard provides an elaboration of the measurement process from ISO/IEC/IEEE 15288:2008 and ISO/IEC/IEEE 12207:2008.

IEEE Std. 15939-2008 Standard Adoption of ISO/IEC 15939:2007 Systems and Software Engineering—Measurement Process

ISO/IEC 15939 defines a measurement process applicable to system and software engineering and management disciplines. The process is described through a model that defines the activities of the measurement process that are required to adequately specify what measurement information is required, how the measures and analysis results are to be applied, and how to determine

if the analysis results are valid. The measurement process is flexible, tailorable, and adaptable to the needs of different users.

ISO/IEC 15939:2007 identifies a process that supports defining a suitable set of measures that address specific information needs. It identifies the activities and tasks that are necessary to successfully identify, define, select, apply, and improve measurement within an overall project or organizational measurement structure. It also provides definitions for measurement terms commonly used within the system and software industries.

Software projects often require the development of user documentation. Management of the project, therefore, includes management of the documentation effort.

ISO/IEC/IEEE 26511:2012 Systems and Software Engineering—Requirements for Managers of User Documentation

ISO/IEC/IEEE 26511:2012 specifies procedures for managing user documentation throughout the software life cycle. It applies to people or organizations producing suites of documentation, to those undertaking a single documentation project, and to documentation produced internally, as well as to documentation contracted to outside service organizations. It provides an overview of the software documentation and information management processes, and also presents aspects of portfolio planning and content management that user documentation managers apply. It covers management activities in starting a project, including setting up procedures and specifications, establishing infrastructure, and building a team. It includes examples of roles needed on a user documentation team. It addresses measurements and estimates needed for management control, and the use of supporting processes such as change management, schedule and cost control, resource management, and quality management and process improvement. It includes requirements for key documents produced for user documentation management, including documentation plans and documentation management plans. ISO/IEC/IEEE 26511:2012 is independent of the software tools that may be used to produce or manage documentation, and applies to both printed documentation and onscreen documentation. Much of its guidance is applicable to user documentation for systems including hardware as well as software.

Sometimes software or system components are acquired rather than developed.

IEEE Std. 1062-1998 Recommended Practice for Software Acquisition

A set of useful quality practices that can be selected and applied during one or more steps in a software acquisition process is described. This recommended practice can be applied to software that runs on any computer system regardless of the size, complexity, or criticality of the software, but is more suited for use on modified-off-the-shelf software and fully developed software.

Sometimes user documentation is acquired regardless of whether the software it describes was acquired. The following standard deals with that subject.

ISO/IEC/IEEE 26512:2011 Systems and Software Engineering—Requirements for Acquirers and Suppliers of User Documentation

ISO/IEC/IEEE 26512:2011 was developed to assist users of ISO/IEC/IEEE 15288:2008 or ISO/IEC/IEEE 12207:2008 to acquire or supply software user documentation as part of the software life cycle processes. It defines the documentation process from the acquirer's standpoint and the supplier's standpoint. ISO/IEC/IEEE 26512:2011 covers the requirements for information items used in the acquisition of user documentation products: the acquisition plan, document specification, statement of work, request for proposals, and proposal. It provides an overview of the software user documentation and information management processes which may require acquisition and supply of software user documentation products and services. It addresses the preparation of requirements for

software user documentation. These requirements are central to the user documentation specification and statement of work. It includes requirements for primary document outputs of the acquisition and supply process: the request for proposal and the proposal for user documentation products and services. It also discusses the use of a documentation management plan and a document plan as they arise in the acquisition and supply processes. ISO/IEC/IEEE 26512:2011 is independent of the software tools that may be used to produce documentation and applies to both printed documentation and onscreen documentation. Much of its guidance is applicable to user documentation for systems including hardware as well as software.

The next two standards are mentioned here because they supply information used in management decision-making.

IEEE Std. 1028-2008 Standard for Software Reviews and Audits

See Software Quality KA

IEEE Std. 1061-1998 Standard for Software Quality Metrics Methodology

See Software Quality KA

The next standard is mentioned because it includes the manager's role in developing user documentation in an agile project.

ISO/IEC/IEEE 26515:2012 Systems and Software Engineering—Developing User Documentation in an Agile Environment

See Software Engineering Models and Methods KA

SOFTWARE ENGINEERING PROCESS

Software and systems engineering processes are central to the standardization of those two disciplines—not just because many are interested in process improvement, but also because processes are effective for the description of improved practices. For example, one might propose an improved practice for software requirements analysis. A naïve treatment might relate the description to an early stage of the life cycle model. A superior approach is to describe the practice in the context of a process that can be applied at any stage of the life cycle. The requirements analysis process, for example, is necessary for the development stage, for maintenance, and often for retirement, so an improved practice described in terms of the requirements analysis process can be applied to any of those stages.

The two key standards are ISO/IEC/IEEE 12207, *Software Life Cycle Processes*, and ISO/IEC/IEEE 15288, *System Life Cycle Processes*. The two standards have distinct histories, but they were both revised in 2008 to align their processes, permitting their interoperable use across a wide spectrum of projects ranging from a standalone software component to a system with negligible software content. Both are being revised again with the intent of containing an identical list of processes, but with provisions specialized for the respective disciplines.

IEEE Std. 12207-2008 (a.k.a. ISO/IEC 12207:2008) Standard for Systems and Software Engineering—Software Life Cycle Processes

ISO/IEC 12207:2008 establishes a common framework for software life cycle processes, with well-defined terminology that can be referenced by the software industry.

ISO/IEC 12207:2008 applies to the acquisition of systems and software products and services and to the supply, development, operation, maintenance, and disposal of software products and the software portion of a system, whether performed internally or externally to an organization. Those aspects of system definition needed to provide the context for software products and services are included.

ISO/IEC 12207:2008 also provides a process that can be employed for defining, controlling, and improving software life cycle processes.

The processes, activities and tasks of ISO/IEC 12207:2008—either alone or in conjunction with ISO/IEC 15288—may also be applied during the acquisition of a system that contains software.

IEEE Std. 15288-2008 (a.k.a. ISO/IEC 15288:2008) Standard for Systems and Software Engineering—System Life Cycle Processes

ISO/IEC 15288:2008 establishes a common framework for describing the life cycle of systems created by humans. It defines a set of processes and associated terminology. These processes can be applied at any level in the hierarchy of a system's structure. Selected sets of these processes can be applied throughout the life cycle for managing and performing the stages of a system's life cycle. This is accomplished through the involvement of all interested parties, with the ultimate goal of achieving customer satisfaction.

ISO/IEC 15288:2008 also provides processes that support the definition, control, and improvement of the life cycle processes used within an organization or a project. Organizations and projects can use these life cycle processes when acquiring and supplying systems.

ISO/IEC 15288:2008 concerns those systems that are man-made and may be configured with one or more of the following: hardware, software, data, humans, processes (e.g., processes for providing service to users), procedures (e.g., operator instructions), facilities, materials, and naturally occurring entities. When a system element is software, the software life cycle processes documented in ISO/IEC 12207:2008 may be used to implement that system element.

ISO/IEC 15288:2008 and ISO/IEC 12207:2008 are harmonized for concurrent use on a single project or in a single organization.

Those two standards specify that processes may produce items of information but do not prescribe their content or format. The next standard provides help with that.

ISO/IEC/IEEE 15289:2011 Systems and Software Engineering—Content of Life-Cycle Information Products (Documentation)

ISO/IEC/IEEE 15289:2011 provides requirements for identifying and planning the specific information items (information products, documentation) to be developed and revised during systems and software life cycles and service management processes. It specifies the purpose and content of all identified systems and software data records and life cycle information items, as well as records and information items for information technology service management. The information item contents are defined according to generic document types (description, plan, policy, procedure, report, request, and specification) and the specific purpose of the document. For simplicity of reference, each information item is described as if it were published as a separate document. However, information items may be unpublished but available in a repository for reference, divided into separate documents or volumes, or combined with other information items into one document. ISO/IEC/IEEE 15289:2011 is based on the life cycle processes specified in ISO/IEC 12207:2008 (IEEE Std. 12207-2008) and ISO/IEC 15288:2008 (IEEE Std. 15288-2008), and the service management processes specified in ISO/IEC 20000-1:2005 and ISO/IEC 20000-2:2005.

The next two guides provide supplementary information helpful in applying 12207 and 15288.

IEEE Std. 24748.2-2012 Guide—Adoption of ISO/IEC TR 24748-2:2011 Systems and Software Engineering—Life Cycle Management—Part 2: Guide to the Application of ISO/IEC 15288 (System Life Cycle Processes)

ISO/IEC TR 24748-2 is a guide for the application of ISO/IEC 15288:2008. It addresses system, life cycle, process, organizational, project, and adaptation concepts, principally through reference to ISO/IEC TR 24748-1 and ISO/IEC 15288:2008. It then gives guidance on applying ISO/IEC 15288:2008 from the aspects of strategy, planning, application in organizations, and application on projects.

IEEE Std. 24748.3-2012 Guide—Adoption of ISO/IEC TR 24748-3:2011 Systems and Software

Engineering—Life Cycle Management—Part 3: Guide to the Application of ISO/IEC 12207 (Software Life Cycle Processes)

ISO/IEC TR 24748-3 is a guide for the application of ISO/IEC 12207:2008. It addresses system, life cycle, process, organizational, project, and adaptation concepts, principally through reference to ISO/IEC TR 24748-1 and ISO/IEC 12207:2008. It gives guidance on applying ISO/IEC 12207:2008 from the aspects of strategy, planning, application in organizations, and application on projects.

The 12207 and 15288 standards provide processes covering the life cycle, but they do not provide a standard life cycle model (waterfall, incremental delivery, prototype-driven, etc). Selecting an appropriate life cycle model for a project is a major concern of ISO/IEC 24748-1.

IEEE Std. 24748.1-2011 Guide—Adoption of ISO/IEC TR 24748-1:2010 Systems and Software Engineering—Life Cycle Management—Part 1: Guide for Life Cycle Management

ISO/IEC TR 24748-1 provides information on life cycle concepts and descriptions of the purposes and outcomes of representative life cycle stages. It also illustrates the use of a life cycle model for systems in the context of ISO/IEC 15288 and provides a corresponding illustration of the use of a life cycle model for software in the context of ISO/IEC 12207. ISO/IEC TR 24748-1 additionally provides detailed discussion and advice on adapting a life cycle model for use in a specific project and organizational environment. It further provides guidance on life cycle model use by domains, disciplines and specialties. ISO/IEC TR 24748-1 gives a detailed comparison between prior and current versions of ISO/IEC 12207 and ISO/IEC 15288 as well as advice on transitioning from prior to current versions and on using their application guides. The discussion and advice are intended to provide a reference model for life cycle models, facilitate use of the updated ISO/IEC 15288 and ISO/IEC 12207, and provide a framework for the development of updated application guides for those International Standards. ISO/IEC TR 24748-1 is a result of the alignment stage of the harmonization of ISO/IEC 12207 and ISO/IEC 15288.

The next standard extends the provisions of ISO/IEC/IEEE 12207 to deal with systematic software reuse.

IEEE Std. 1517-2010 Standard for Information Technology—System and Software Life Cycle Processes—Reuse Processes

A common framework for extending the system and software life cycle processes of IEEE Std. 12207:2008 to include the systematic practice of reuse is provided. The processes, activities, and tasks to be applied during each life cycle process to enable a system and/or product to be constructed from reusable assets are specified. The processes, activities, and tasks to enable the identification, construction, maintenance, and management of assets supplied are also specified.

IEEE Std. 1220 has been widely applied as a systems engineering process and was adopted by ISO/IEC with the number 26702. Unfortunately, the standard is not completely compatible with ISO/IEC/IEEE 15288 and is being revised to solve that problem. The result will be published as ISO/IEC/IEEE 24748-4.

IEEE Std. 1220-2005 (a.k.a. ISO/IEC 26702:2007) Standard for Application and Management of the Systems Engineering Process

ISO/IEC 26702 defines the interdisciplinary tasks which are required throughout a system's life cycle to transform customer needs, requirements, and constraints into a system solution. In addition, it specifies the requirements for the systems engineering process and its application throughout the product life cycle. ISO/IEC 26702:2007 focuses on engineering activities necessary to guide product development, while ensuring

that the product is properly designed to make it affordable to produce, own, operate, maintain, and eventually dispose of without undue risk to health or the environment.

Since SC 7 and IEEE have written so many process standards, one may not be surprised to learn that their model for process description is recorded in a Technical Report.

IEEE Std. 24774-2012 Guide—Adoption of ISO/IEC TR 24474:2010 Systems and Software Engineering—Life Cycle Management—Guidelines for Process Description

An increasing number of international, national, and industry standards describe process models. These models are developed for a range of purposes including process implementation and assessment. The terms and descriptions used in such models vary in format, content, and level of prescription. ISO/IEC TR 24774:2010 presents guidelines for the elements used most frequently in describing a process: the title, purpose, outcomes, activities, task, and information item. Whilst the primary purpose of ISO/IEC TR 24774:2010 is to encourage consistency in standard process reference models, the guidelines it provides can be applied to any process model developed for any purpose.

A very small entity (VSE) is an enterprise, an organization, a department, or a project having up to 25 people. The ISO/IEC 29110 series "profiles" large standards, such as ISO/IEC 12207 for software and ISO/IEC 15288 for systems, into smaller ones for VSEs. ISO 29110 is applicable to VSEs that do not develop critical systems or critical software. Profiles provide a roadmap allowing a start-up to grow a step at a time using the ISO 29110 management and engineering guides.

ISO/IEC 29110 set of standards and technical reports are targeted by audience such as VSEs, customers, or auditors. ISO/IEC 29110 is not intended to preclude the use of different life cycles approaches such as waterfall, iterative, incremental, evolutionary, or agile.

A VSE could obtain an ISO/IEC 29110 Certification. The set of technical reports is available at no cost on the ISO website. Many ISO 29110 documents are available in English, Spanish, Portuguese, Japanese, and French.

ISO/IEC TR 29110-5-1-2:2011 Software Engineering—Lifecycle Profiles for Very Small Entities (VSEs)—Part 5-1-2: Management and Engineering Guide: Generic Profile Group: Basic Profile

ISO/IEC TR 29110-5-1-2:2011 is applicable to very small entities (VSEs). A VSE is defined as an enterprise, organization, department, or project having up to 25 people. A set of standards and guides has been developed according to a set of VSEs' characteristics and needs. The guides are based on subsets of appropriate standards elements, referred to as VSE profiles. The purpose of a VSE profile is to define a subset of ISO/IEC international standards relevant to the VSEs' context.

ISO/IEC TR 29110-5-1-2:2011 provides the management and engineering guide to the basic VSE profile applicable to VSEs that do not develop critical software. The generic profile group does not imply any specific application domain.

The next standard may be viewed as an alternative to 12207 for individual projects. The 1074 standard explains how to define processes for use on a given project. The 12207 and 15288 standards, however, focus on defining processes for organizational adoption and repeated use on many projects. The current 1074 is the update of a standard that was a predecessor of 12207.

IEEE Std. 1074-2006 Standard for Developing a Software Project Life Cycle Process

This standard provides a process for creating a software project life cycle process (SPLCP). It is primarily directed at the process architect for a given software project.

All of the standards described so far in this section provide a basis for *defining* processes. Some users are interested in *assessing* and improving their processes after implementation. The 15504 series provides for process assessment; it is currently being revised and renumbered 330xx.

ISO/IEC 15504 [ten parts] Information Technology—Process Assessment

ISO/IEC 15504-2:2003 defines the requirements for performing process assessment as a basis for use in process improvement and capability determination.

Process assessment is based on a two-dimensional model containing a process dimension and a capability dimension. The process dimension is provided by an external process reference model (such as 12207 or 15288), which defines a set of processes characterized by statements of process purpose and process outcomes. The capability dimension consists of a measurement framework comprising six process capability levels and their associated process attributes.

The assessment output consists of a set of process attribute ratings for each process assessed, termed the process profile, and may also include the capability level achieved by that process.

ISO/IEC 15504-2:2003 identifies the measurement framework for process capability and the requirements for

- performing an assessment;
- process reference models;
- process assessment models;
- verifying conformity of process assessment.

The requirements for process assessment defined in ISO/IEC 15504-2:2003 form a structure that

- facilitates self-assessment;
- provides a basis for use in process improvement and capability determination;
- takes into account the context in which the assessed process is implemented;
- produces a process rating;
- addresses the ability of the process to achieve its purpose;
- is applicable across all application domains and sizes of organization; and
- may provide an objective benchmark between organizations.

The minimum set of requirements defined in ISO/IEC 15504-2:2003 ensures that assessment results are objective, impartial, consistent, repeatable, and representative of the assessed processes. Results of conformant process assessments may be compared when the scopes of the assessments are considered to be similar; for guidance on this matter, refer to ISO/IEC 15504-4.

Several other standards are mentioned here because they are written as elaborations of the processes of 12207 or 15288. They are allocated to other KAs because each one deals with topics described in those other KAs.

IEEE Std. 828-2012 Standard for Configuration Management in Systems and Software Engineering

See Software Configuration Management KA

IEEE Std. 14764-2006 (a.k.a. ISO/IEC 14764:2006) Standard for Software Engineering—Software Life Cycle Processes—Maintenance

See Software Maintenance KA

ISO/IEC 15026-4:2012 Systems and Software Engineering—Systems and Software Assurance—Part 4: Assurance in the Life Cycle

See Software Quality KA

IEEE Std. 15939-2008 Standard Adoption of ISO/IEC 15939:2007 Systems and Software Engineering—Measurement Process

See Software Engineering Management KA

ISO/IEC 15940:2006 Information Technology—Software Engineering Environment Services

See Software Engineering Models and Methods KA

IEEE Std. 16085-2006 (a.k.a. ISO/IEC 16085:2006) Standard for Systems and Software Engineering—Software Life Cycle Processes—Risk Management

See Software Engineering Management KA

ISO/IEC/IEEE 16326:2009 Systems and Software Engineering—Life Cycle Processes—Project Management

See Software Engineering Management KA

ISO/IEC/IEEE 29148:2011 Systems and Software Engineering—Life Cycle Processes—Requirements Engineering

See Software Requirements KA

Some users desire process standards usable for IT operations or IT service management. The ISO/IEC 20000 series describe IT service management. The processes are less rigorously defined than those of the aforementioned engineering standards, but may be preferable for situations where the risks of failure involve money or customer satisfaction rather than public health, safety, and welfare. The ISO/IEC 20000 series now extend to many parts. The foundation of the series, ISO/IEC 20000-1, is briefly described below.

ISO/IEC 20000-1:2011 Information Technology—Service Management—Part 1: Service Management System Requirements

ISO/IEC 20000-1:2011 is a service management system (SMS) standard. It specifies requirements for the service provider to plan, establish, implement, operate, monitor, review, maintain, and improve an SMS. The requirements include the design, transition, delivery and improvement of services to fulfill agreed service requirements.

IEEE has adopted the first two parts of the ISO/IEC 20000 series.

SOFTWARE ENGINEERING MODELS AND METHODS

Some approaches to software engineering use methods that cut across large parts of the life cycle, rather than focusing on specific processes. "Chief Programmer" was one traditional example. "Agile development" (actually an example of traditional incremental delivery) is a current example. Neither S2ESC nor SC 7 has a standard for agile development, but there is a standard for developing user documentation in an agile project.

ISO/IEC/IEEE 26515:2012 Systems and Software Engineering—Developing User Documentation in an Agile Environment

ISO/IEC/IEEE 26515:2012 specifies the way in which user documentation can be developed in agile development projects. It is intended for use in all organizations that are using agile development or are considering implementing their projects using these techniques. It applies to people or organizations producing suites of documentation, to those undertaking a single documentation project, and to documentation produced internally, as well as to documentation contracted to outside service organizations. ISO/IEC/IEEE 26515:2012 addresses the relationship between the user documentation process and the life cycle documentation process in agile development. It describes how the information developer or project manager may plan and manage the user documentation development in an agile environment. It is intended neither to encourage nor to discourage the use of any particular agile development tools or methods.

Many methodologies are based on semiformal descriptions of the software to be constructed. These range from simple descriptive notations to models that can be manipulated and tested and, in some cases, can generate code. Two relatively old techniques start the list; the first has been widely applied for modeling processes and workflows.

IEEE Std. 1320.1-1998 Standard for Functional Modeling Language—Syntax and Semantics for IDEF0

IDEF0 function modeling is designed to represent the decisions, actions, and activities of an existing or prospective organization or system. IDEF0 graphics and accompanying texts are presented in an organized and systematic way to gain

understanding, support analysis, provide logic for potential changes, specify requirements, and support system-level design and integration activities. IDEF0 may be used to model a wide variety of systems, composed of people, machines, materials, computers, and information of all varieties, and structured by the relationships among them, both automated and nonautomated. For new systems, IDEF0 may be used first to define requirements and to specify the functions to be carried out by the future system. As the basis of this architecture, IDEF0 may then be used to design an implementation that meets these requirements and performs these functions. For existing systems, IDEF0 can be used to analyze the functions that the system performs and to record the means by which these are done.

IEEE Std. 1320.2-1998 Standard for Conceptual Modeling Language—Syntax and Semantics for IDEF1X97 (IDEFobject)

IDEF1X 97 consists of two conceptual modeling languages. The key-style language supports data/information modeling and is downward compatible with the US government's 1993 standard, FIPS PUB 184. The identity-style language is based on the object model with declarative rules and constraints. IDEF1X 97 identity style includes constructs for the distinct but related components of object abstraction: interface, requests, and realization; utilizes graphics to state the interface; and defines a declarative, directly executable rule and constraint language for requests and realizations. IDEF1X 97 conceptual modeling supports implementation by relational databases, extended relational databases, object databases, and object programming languages. IDEF1X 97 is formally defined in terms of first order logic. A procedure is given whereby any valid IDEF1X 97 model can be transformed into an equivalent theory in first order logic. That procedure is then applied to a metamodel of IDEF1X 97 to define the valid set of IDEF1X 97 models.

In recent years, the UML notation has become popular for modeling software-intensive systems. The next two standards provide two versions of the UML language.

ISO/IEC 19501:2005 Information Technology—Open Distributed Processing—Unified Modeling Language (UML) Version 1.4.2

ISO/IEC 19501 describes the Unified Modeling Language (UML), a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components.

ISO/IEC 19505:2012 [two parts] Information Technology—Object Management Group Unified Modeling Language (OMG UML)

ISO/IEC 19505 defines the Unified Modeling Language (UML), revision 2. The objective of UML is to provide system architects, software engineers, and software developers with tools for analysis, design, and implementation of software-based systems as well as for modeling business and similar processes.

Two more standards build on the base of UML to provide additional modeling capabilities:

ISO/IEC 19506:2012 Information Technology—Object Management Group Architecture-Driven Modernization (ADM)—Knowledge Discovery Meta-Model (KDM)

ISO/IEC 19506:2012 defines a metamodel for representing existing software assets, their associations, and operational environments, referred to as the knowledge discovery metamodel (KDM). This is the first in the series of specifications related to software assurance (SwA) and architecture-driven modernization (ADM) activities. KDM facilitates

projects that involve existing software systems by insuring interoperability and exchange of data between tools provided by different vendors.

ISO/IEC 19507:2012 Information Technology—Object Management Group Object Constraint Language (OCL)

ISO/IEC 19507:2012 defines the Object Constraint Language (OCL), version 2.3.1. OCL version 2.3.1 is the version of OCL that is aligned with UML 2.3 and MOF 2.0.

Some organizations invest in software engineering environments (SEE) to assist in the construction of software. An SEE, per se, is not a replacement for sound processes. However, a suitable SEE must support the processes that have been chosen by the organization.

ISO/IEC 15940:2006 Information Technology—Software Engineering Environment Services

ISO/IEC 15940:2006 defines software engineering environment (SEE) services conceptually in a reference model that can be adapted to any SEEs to automate one or more software engineering activities. It describes services that support the process definitions as in ISO/IEC 12207 so that the set of SEE services is compatible with ISO/IEC 12207. ISO/IEC 15940:2006 can be used either as a general reference or to define an automated software process.

The selection of tooling for a software engineering environment is itself a difficult task. Two standards provide some assistance. ISO/IEC 14102:2008 defines both a set of processes and a structured set of computer-aided software engineering (CASE) tool characteristics for use in the technical evaluation and the ultimate selection of a CASE tool.

IEEE Std. 14102-2010 Standard Adoption of ISO/IEC 14102:2008 Information Technology—Guideline for the Evaluation and Selection of CASE Tools

Within systems and software engineering, computer-aided software engineering (CASE) tools represent a major part of the supporting technologies used to develop and maintain information technology systems. Their selection must be carried out with careful consideration of both the technical and management requirements.

ISO/IEC 14102:2008 defines both a set of processes and a structured set of CASE tool characteristics for use in the technical evaluation and the ultimate selection of a CASE tool. It follows the software product evaluation model defined in ISO/IEC 14598-5:1998.

ISO/IEC 14102:2008 adopts the general model of software product quality characteristics and subcharacteristics defined in ISO/IEC 9126-1:2001 and extends these when the software product is a CASE tool; it provides product characteristics unique to CASE tools.

The next document provides guidance on how to adopt CASE tools, once selected.

IEEE Std. 14471-2010 Guide—Adoption of ISO/IEC TR 14471:2007 Information Technology—Software Engineering—Guidelines for the Adoption of CASE Tools

The purpose of ISO/IEC TR 14471:2007 is to provide a recommended practice for CASE adoption. It provides guidance in establishing processes and activities that are to be applied for the successful adoption of CASE technology. The use of ISO/IEC TR 14471:2007 will help to maximize the return and minimize the risk of investing in CASE technology. However, ISO/IEC TR 14471:2007 does not establish compliance criteria.

It is best used in conjunction with ISO/IEC 14102 for CASE tool evaluation and selection. It neither dictates nor advocates particular development standards, software processes, design methods, methodologies, techniques, programming languages, or life cycle paradigms.

Within a software engineering environment, it is important for the various tools to interoperate. The following standards provide a scheme for interconnection.

IEEE Std. 1175.1-2002 Guide for CASE Tool Interconnections—Classification and Description

IEEE Std. 1175.2-2006 Recommended Practice for CASE Tool Interconnection—Characterization of Interconnections

IEEE Std. 1175.3-2004 Standard for CASE Tool Interconnections—Reference Model for Specifying Software Behavior

IEEE Std. 1175.4-2008 Standard for CASE Tool Interconnections—Reference Model for Specifying System Behavior

The purpose of this family of standards is to specify a common set of modeling concepts based on those found in commercial CASE tools for describing the operational behavior of a software system. These standards establish a uniform, integrated model of software concepts related to software functionality. They also provide a textual syntax for expressing the common properties (attributes and relationships) of those concepts as they have been used to model software behavior.

SOFTWARE QUALITY

One viewpoint of software quality starts with ISO 9001, *Quality Management Requirements*, dealing with quality policy throughout an organization. The terminology of that standard may be unfamiliar to software professionals, and quality management auditors may be unfamiliar with software jargon. The following standard describes the relationship between ISO 9001 and ISO/IEC 12207. Unfortunately, the current version refers to obsolete editions of both; a replacement is in progress:

IEEE Std. 90003-2008 Guide—Adoption of ISO/IEC 90003:2004 Software Engineering—Guidelines for the Application of ISO 9001:2000 to Computer Software

ISO/IEC 90003 provides guidance for organizations in the application of ISO 9001:2000 to the acquisition, supply, development, operation, and maintenance of computer software and related support services. ISO/IEC 90003:2004 does not add to or otherwise change the requirements of ISO 9001:2000.

The guidelines provided in ISO/IEC 90003:2004 are not intended to be used as assessment criteria in quality management system registration/certification.

The application of ISO/IEC 90003:2004 is appropriate to software that is

- part of a commercial contract with another organization,
- a product available for a market sector,
- used to support the processes of an organization,
- embedded in a hardware product, or
- related to software services.

Some organizations may be involved in all the above activities; others may specialize in one area. Whatever the situation, the organization's quality management system should cover all aspects (software related and nonsoftware related) of the business.

ISO/IEC 90003:2004 identifies the issues which should be addressed and is independent of the technology, life cycle models, development processes, sequence of activities, and organizational structure used by an organization. Additional guidance and frequent references to the ISO/IEC JTC 1/SC 7 software engineering standards are provided to assist in the application of ISO 9001:2000: in particular, ISO/IEC 12207, ISO/IEC TR 9126, ISO/IEC 14598, ISO/IEC 15939, and ISO/IEC TR 15504.

The ISO 9001 approach posits an organization-level quality management process paired with project-level quality assurance planning to achieve the organizational goals. IEEE 730 describes project-level quality planning. It is

currently aligned with an obsolete edition of 12207, but a revision is being prepared.

IEEE Std. 730-2002 Standard for Software Quality Assurance Plans

The standard specifies the format and content of software quality assurance plans.

Another viewpoint of software quality begins with enumerating the desired characteristics of a software product and selecting measures or other evaluations to determine if the desired level of characteristics has been achieved. The so-called SQuaRE (software product quality requirements and evaluation) series of SC 7 standards covers this approach in great detail.

ISO/IEC 25000 through 25099 Software Engineering—Software Product Quality Requirements and Evaluation (SQuaRE)

A few of the SQuaRE standards are selected below for particular attention. The first is the overall guide to the series.

ISO/IEC 25000:2005 Software Engineering—Software Product Quality Requirements and Evaluation (SQuaRE)—Guide to SQuaRE

ISO/IEC 25000:2005 provides guidance for the use of the new series of international standards named Software product Quality Requirements and Evaluation (SQuaRE). The purpose of this guide is to provide a general overview of SQuaRE contents, common reference models, and definitions, as well as the relationship among the documents, allowing users of this guide a good understanding of those international standards. This document contains an explanation of the transition process between the old ISO/IEC 9126 and the 14598 series and SQuaRE, and also presents information on how to use the ISO/IEC 9126 and 14598 series in their previous form.

SQuaRE provides

- terms and definitions,
- reference models,
- guides
- standards for requirements specification, planning and management, measurement, and evaluation purposes.

The next SQuaRE standard provides a taxonomy of software quality characteristics that may be useful in selecting characteristics relevant to a specific project:

ISO/IEC 25010:2011 Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—System and Software Quality Models

ISO/IEC 25010:2011 defines the following:

1. A quality in-use model composed of five characteristics (some of which are further subdivided into subcharacteristics) that relate to the outcome of interaction when a product is used in a particular context of use. This system model is applicable to the complete human-computer system, including both computer systems in use and software products in use.
2. A product quality model composed of eight characteristics (which are further subdivided into subcharacteristics) that relate to static properties of software and dynamic properties of the computer system. The model is applicable to both computer systems and software products.

The characteristics defined by both models are relevant to all software products and computer systems. The characteristics and subcharacteristics provide consistent terminology for specifying, measuring, and evaluating system and software product quality. They also provide a set of quality characteristics against which stated quality requirements can be compared for completeness.

Although the scope of the product quality model is intended to be software and computer systems, many of the characteristics are also relevant to wider systems and services.

ISO/IEC 25012 contains a model for data quality that is complementary to this model.

The scope of the models excludes purely functional properties, but it does include functional suitability.

The scope of application of the quality models includes supporting specification and evaluation of software and software-intensive computer systems from different perspectives by those who are associated with their acquisition, requirements, development, use, evaluation, support, maintenance, quality assurance and control, and audit. The models can, for example, be used by developers, acquirers, quality assurance and control staff, and independent evaluators, particularly those responsible for specifying and evaluating software product quality. Activities during product development that can benefit from the use of the quality models include

- identifying software and system requirements;
- validating the comprehensiveness of a requirements definition;
- identifying software and system design objectives;
- identifying software and system testing objectives;
- identifying quality control criteria as part of quality assurance;
- identifying acceptance criteria for a software product and/or software-intensive computer system;
- establishing measures of quality characteristics in support of these activities.

Some documents in the SQuaRE series deal specifically with the characteristic of usability. The Common Industry Format (CIF) for usability reporting began at the US National Institute for Standards and Technology (NIST) and was moved into ISO/IEC JTC 1/SC 7 for purposes of standardization.

ISO/IEC 25060 through 25064 Software Engineering—Software Product Quality Requirements and Evaluation (SQuaRE)—Common Industry Format (CIF) for Usability

A family of international standards, named the Common Industry Formats (CIF), documents the specification and evaluation of the usability of interactive systems. It provides a general overview of the CIF framework and contents, definitions, and the relationship of the framework elements. The intended users of the framework are identified, as well as the situations in which the framework may be applied. The assumptions and constraints of the framework are also enumerated.

The framework content includes the following:

- consistent terminology and classification of specification, evaluation, and reporting;
- a definition of the type and scope of formats and the high-level structure to be used for documenting required information and the results of evaluation.

The CIF family of standards is applicable to software and hardware products used for predefined tasks. The information items are intended to be used as part of system-level documentation resulting from development processes such as those in ISO 9241-210 and ISO/IEC JTC 1/SC 7 process standards.

The CIF family focuses on documenting those elements needed for design and development of usable systems, rather than prescribing a specific process. It is intended to be used in conjunction with existing international standards, including ISO 9241, ISO 20282, ISO/IEC 9126, and the SQuaRE series (ISO/IEC 25000 to ISO/IEC 25099).

The CIF family of standards does not prescribe any kind of method, life cycle or process.

Not everyone agrees with the taxonomy of quality characteristics in ISO/IEC 25010. That standard has a quality factor called "reliability" that has subfactors of maturity, availability, fault tolerance, and recoverability. IEC TC 65, which has responsibility for standards on "dependability," defines that term as a nonquantitative composite of reliability, maintainability, and maintenance support. Others use the term "reliability"

to denote a measure defined by a mathematical equation. The disagreement over the use of these words means that the standards on the subject are inherently unaligned. A few will be noted below, but the words like those noted above may mean different things in different standards.

IEEE Std. 982.1-2005 Standard for Dictionary of Measures of the Software Aspects of Dependability

A standard dictionary of measures of the software aspects of dependability for assessing and predicting the reliability, maintainability, and availability of any software system; in particular, it applies to mission critical software systems.

IEEE Std. 1633-2008 Recommended Practice for Software Reliability

The methods for assessing and predicting the reliability of software, based on a life cycle approach to software reliability engineering, are prescribed in this recommended practice. It provides information necessary for the application of software reliability (SR) measurement to a project, lays a foundation for building consistent methods, and establishes the basic principle for collecting the data needed to assess and predict the reliability of software. The recommended practice prescribes how any user can participate in SR assessments and predictions.

IEEE has an overall standard for software product quality that has a scope similar to the ISO/IEC 250xx series described previously. Its terminology differs from the ISO/IEC series, but it is substantially more compact.

IEEE Std. 1061-1998 Standard for Software Quality Metrics Methodology

A methodology for establishing quality requirements and identifying, implementing, analyzing, and validating the process and product software quality metrics is defined. The methodology spans the entire software life cycle.

One approach to achieving software quality is to perform an extensive program of verification and validation. IEEE Std. 1012 is probably the world's most widely applied standard on this subject. A revision was recently published.

IEEE Std. 1012-2012 Standard for System and Software Verification and Validation

Verification and validation (V&V) processes are used to determine whether the development products of a given activity conform to the requirements of that activity and whether the product satisfies its intended use and user needs. V&V life cycle process requirements are specified for different integrity levels. The scope of V&V processes encompasses systems, software, and hardware, and it includes their interfaces. This standard applies to systems, software, and hardware being developed, maintained, or reused [legacy, commercial off-the-shelf (COTS), nondevelopmental items]. The term software also includes firmware and microcode, and each of the terms system, software, and hardware includes documentation. V&V processes include the analysis, evaluation, review, inspection, assessment, and testing of products.

There are other standards that support the verification and validation processes. One describes techniques for performing reviews and audits during a software project.

IEEE Std. 1028-2008 Standard for Software Reviews and Audits

Five types of software reviews and audits, together with procedures required for the execution of each type, are defined in this standard. This standard is concerned only with the reviews and audits; procedures for determining the necessity of a review or audit are not defined, and the disposition of the results of the review or audit is not specified. Types included are management reviews, technical reviews, inspections, walkthroughs, and audits.

In many cases, a database of software anomalies is used to support verification and validation activities. The following standard suggests how anomalies should be classified.

IEEE Std. 1044-2009 Standard for Classification for Software Anomalies

This standard provides a uniform approach to the classification of software anomalies, regardless of when they originate or when they are encountered within the project, product, or system life cycle. Classification data can be used for a variety of purposes, including defect causal analysis, project management, and software process improvement (e.g., to reduce the likelihood of defect insertion and/or increase the likelihood of early defect detection).

In some systems, one particular property of the software is so important that it requires special treatment beyond that provided by a conventional verification and validation program. The emerging term for this sort of treatment is "systems and software assurance." Examples include safety, privacy, high security, and ultrareliability. The 15026 standard is under development to deal with such situations. The first part of the four-part standard provides terminology and concepts used in the remaining parts. It was first written before the other parts and is now being revised for complete agreement with the others.

IEEE Std. 15026.1-2011 Trial-Use Standard Adoption of ISO/IEC TR 15026-1:2010 Systems and Software Engineering—Systems and Software Assurance—Part 1: Concepts and Vocabulary

This trial-use standard adopts ISO/IEC TR 15026-1:2010, which defines terms and establishes an extensive and organized set of concepts and their relationships for software and systems assurance, thereby establishing a basis for shared understanding of the concepts and principles central to ISO/IEC 15026 across its user communities. It provides information to users of the subsequent parts of ISO/IEC 15026, including the use of each part and the combined use of multiple parts. Coverage of assurance for a service being operated and managed on an ongoing basis is not covered in ISO/IEC 15026.

The second part of the standard describes the structure of an "assurance case," which is intended as a structured argument that the critical property has been achieved. It is a generalization of various domain-specific constructs like "safety cases."

IEEE Std. 15026.2-2011 Standard Adoption of ISO/IEC 15026-2:2011 Systems and Software Engineering—Systems and Software Assurance—Part 2: Assurance Case

ISO/IEC 15026-2:2011 is adopted by this standard. ISO/IEC 15026-2:2011 specifies minimum requirements for the structure and contents of an assurance case to improve the consistency and comparability of assurance cases and to facilitate stakeholder communications, engineering decisions, and other uses of assurance cases. An assurance case includes a top-level claim for a property of a system or product (or set of claims), systematic argumentation regarding this claim, and the evidence and explicit assumptions that underlie this argumentation. Arguing through multiple levels of subordinate claims, this structured argumentation connects the top-level claim to the evidence and assumptions. Assurance cases are generally developed to support claims in areas such as safety, reliability, maintainability, human factors, operability, and security, although these assurance cases are often called by more specific names, e.g., safety case or reliability and maintainability (R&M) case. ISO/IEC 15026-2:2011 does not place requirements on the quality of the contents of an assurance case and does not require the use of a particular terminology or graphical representation. Likewise, it places no requirements on the means of physical implementation of the data, including no requirements for redundancy or colocation.

In many systems, some portions are critical to achieving the desired property while others are only

incidental. For example, the flight control system of an airliner is critical to safety, but the microwave oven is not. Conventionally, the various portions are assigned "criticality levels" to indicate their significance to the overall achievement of the property. The third part of ISO/IEC 15026 describes how that is done. This part will be revised for better fit with the remainder of the 15026 standard.

ISO/IEC 15026-3:2011 Systems and Software Engineering—Systems and Software Assurance—Part 3: System Integrity Levels

ISO/IEC 15026-3:2011 specifies the concept of integrity levels with corresponding integrity level requirements that are required to be met in order to show the achievement of the integrity level. It places requirements on and recommends methods for defining and using integrity levels and their integrity level requirements, including the assignment of integrity levels to systems, software products, their elements, and relevant external dependences.

ISO/IEC 15026-3:2011 is applicable to systems and software and is intended for use by:

- definers of integrity levels such as industry and professional organizations, standards organizations, and government agencies;
- users of integrity levels such as developers and maintainers, suppliers and acquirers, users, and assessors of systems or software, and for the administrative and technical support of systems and/or software products.

One important use of integrity levels is by suppliers and acquirers in agreements; for example, to aid in assuring safety, economic, or security characteristics of a delivered system or product.

ISO/IEC 15026-3:2011 does not prescribe a specific set of integrity levels or their integrity level requirements. In addition, it does not prescribe the way in which integrity level use is integrated with the overall system or software engineering life cycle processes.

ISO/IEC 15026-3:2011 can be used alone or with other parts of ISO/IEC 15026. It can be used with a variety of technical and specialized risk analysis and development approaches. ISO/IEC TR 15026-1 provides additional information and references to aid users of ISO/IEC 15026-3:2011.

ISO/IEC 15026-3:2011 does not require the use of the assurance cases described by ISO/IEC 15026-2 but describes how integrity levels and assurance cases can work together, especially in the definition of specifications for integrity levels or by using integrity levels within a portion of an assurance case.

The final part of 15026 provides additional guidance for executing the life cycle processes of 12207 and 15288 when a system or software is required to achieve an important property.

ISO/IEC 15026-4:2012 Systems and Software Engineering—Systems and Software Assurance—Part 4: Assurance in the Life Cycle

This part of ISO/IEC 15026 gives guidance and recommendations for conducting selected processes, activities and tasks for systems and software products requiring assurance claims for properties selected for special attention, called critical properties. This part of ISO/IEC 15026 specifies a property-independent list of processes, activities, and tasks to achieve the claim and show the achievement of the claim. This part of ISO/IEC 15026 establishes the processes, activities, tasks, guidance, and recommendations in the context of a defined life cycle model and set of life cycle processes for system and/or software life cycle management.

The next standard deals with a property—safety—that is often identified as critical. It was originally developed in cooperation with the US nuclear power industry.

IEEE Std. 1228-1994 Standard for Software Safety Plans

The minimum acceptable requirements for the content of a software safety plan are established. This standard applies to the software safety plan used for the development, procurement, maintenance, and retirement of safety-critical software.

This standard requires that the plan be prepared within the context of the system safety program. Only the safety aspects of the software are included. This standard does not contain special provisions required for software used in distributed systems or in parallel processors.

Classical treatments suggest that "verification" deals with static evaluation methods and that "testing" deals with dynamic evaluation methods. Recent treatments, including ISO/IEC draft 29119, are blurring this distinction, though, so testing standards are mentioned here.

IEEE Std. 829-2008 Standard for Software and System Test Documentation

See Software Testing KA

IEEE Std. 1008-1987 Standard for Software Unit Testing

See Software Testing KA

IEEE Std. 26513-2010 Standard Adoption of ISO/IEC 26513:2009 Systems and Software Engineering—Requirements for Testers and Reviewers of Documentation

See Software Testing KA

ISO/IEC/IEEE 29119 [four parts] (Draft) Software and Systems Engineering—Software Testing

See Software Testing KA

SOFTWARE ENGINEERING PROFESSIONAL PRACTICE

IEEE is a provider of products related to the certification of professional practitioners of software engineering. The first has already been described, the *Guide to the Software Engineering Body of Knowledge*. The *SWEBOK Guide* has been adopted by ISO/IEC as an outline of the knowledge that professional software engineers should have.

ISO/IEC TR 19759:2005 Software Engineering—Guide to the Software Engineering Body of Knowledge (SWEBOK)

See General

An SC 7 standard provides a framework for comparisons among certifications of software engineering professionals. That standard states that the areas considered in certification must be mapped to the *SWEBOK Guide*.

ISO/IEC 24773:2008 Software Engineering—Certification of Software Engineering Professionals

ISO/IEC 24773:2008 establishes a framework for comparison of schemes for certifying software engineering professionals. A certification scheme is a set of certification requirements for software engineering professionals. ISO/IEC 24773:2008 specifies the items that a scheme is required to contain and indicates what should be defined for each item.

ISO/IEC 24773:2008 will facilitate the portability of software engineering professional certifications between different countries or organizations. At present, different countries and organizations have adopted different approaches on the topic, which are implemented by means of regulations and bylaws. The intention of ISO/IEC 24773:2008 is to be open to these individual approaches by providing a framework for expressing them in a common scheme that can lead to understanding.

SC 7 is currently drafting a guide that will supplement 24773.

SOFTWARE ENGINEERING ECONOMICS

No standards are allocated to this KA.

COMPUTING FOUNDATIONS

No standards are allocated to this KA.

MATHEMATICAL FOUNDATIONS

No standards are allocated to this KA.

ENGINEERING FOUNDATIONS

No standards are allocated to this KA.

STAYING CURRENT

This article was obsolescent the moment it was drafted. Some readers will need to know how to get current designations and descriptions of standards. This section describes some helpful resources.

WHERE TO FIND STANDARDS

The list of standards published for ISO/IEC JTC 1/SC 7 can be found at www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_tc_browse.htm?commid=45086.

Because the URL might change, readers might have to navigate to the list. Begin at www.iso.org/iso/store.htm, then click on "browse standards catalogue," then "browse by TC," then "JTC 1," then "SC 7."

Finding the current list of standards for S2ESC is a bit more difficult. Begin at http://standards.ieee.org/. In the search box under "Find Standards," type "S2ESC." This should produce a list of published standards for which S2ESC is responsible.

Keep in mind that the searchable databases are compilations. Like any such database, they can contain errors that lead to incomplete search results.

WHERE TO OBTAIN THE STANDARDS

Some readers will want to obtain standards described in this article. The first thing to know is that some international standards are available free for individual use. The current list of ISO/IEC standards available under these terms is located at http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html.

One of the publicly available standards is the ISO/IEC adoption of the *SWEBOK Guide*, ISO/IEC 19759.

The definitions contained in ISO/IEC/IEEE 24765, *System and Software Vocabulary*, are freely available at www.computer.org/sevocab.

However, the vast majority of standards are not free. ISO/IEC standards are generally purchased from the national standards organization of the country in which one lives. For example, in the US, international standards can be purchased from the American National Standards Institute at http://webstore.ansi.org/. Alternatively, standards can be purchased directly from ISO/IEC at www.iso.org/iso/store.htm. It should be noted that each individual nation is free to set its own prices, so it may be helpful to check both sources.

IEEE standards may be available to you for free if your employer or library has a subscription to IEEE Xplore: http://ieeexplore.ieee.org/. Some subscriptions to Xplore provide access only to the abstracts of standards; the full text may then be purchased via Xplore. Alternatively, standards may be purchased via the IEEE standards store at www.techstreet.com/ieeegate.html. It should be noted that IEEE-SA sometimes bundles standards into groups available at a substantial discount.

Finally, the reader should note that standards that IEEE has adopted from ISO/IEC, standards that ISO/IEC has "fast-tracked" from IEEE, and standards that were jointly developed or revised are available from both sources. For all standards described in this article, the IEEE version and the ISO/IEC version are substantively identical. The respective versions may have different front and back matter but the bodies are identical.

WHERE TO SEE THE SWEBOK GUIDE

The *SWEBOK Guide* is published under an IEEE copyright. The current version of the *SWEBOK Guide* is available free to the public at www.swebok.org/. The ISO/IEC adoption of the *SWEBOK Guide*, ISO/IEC TR 19759, is one of the freely available standards.

SUMMARY LIST OF THE STANDARDS

Number and Title (listed in order of number)	Most Relevant KA
IEEE Std. 730-2002 Standard for Software Quality Assurance Plans	SW Quality
IEEE Std. 828-2012 Standard for Configuration Management in Systems and Software Engineering	SW Configuration Management
IEEE Std. 829-2008 Standard for Software and System Test Documentation	SW Testing
IEEE Std. 982.1-2005 Standard for Dictionary of Measures of the Software Aspects of Dependability	SW Quality
IEEE Std. 1008-1987 Standard for Software Unit Testing	SW Testing
IEEE Std. 1012-2012 Standard for System and Software Verification and Validation	SW Quality
IEEE Std. 1016-2009 Standard for Information Technology—Systems Design—Software Design Descriptions	SW Design
IEEE Std. 1028-2008 Standard for Software Reviews and Audits	SW Quality
IEEE Std. 1044-2009 Standard for Classification for Software Anomalies	SW Quality
IEEE Std. 1061-1998 Standard for Software Quality Metrics Methodology	SW Quality
IEEE Std. 1062-1998 Recommended Practice for Software Acquisition	SW Engineering Management
IEEE Std. 1074-2006 Standard for Developing a Software Project Life Cycle Process	SW Engineering Process
IEEE Std. 1175.1-2002 Guide for CASE Tool Interconnections—Classification and Description	SW Engineering Models and Methods
IEEE Std. 1175.2-2006 Recommended Practice for CASE Tool Interconnection—Characterization of Interconnections	SW Engineering Models and Methods
IEEE Std. 1175.3-2004 Standard for CASE Tool Interconnections—Reference Model for Specifying Software Behavior	SW Engineering Models and Methods
IEEE Std. 1175.4-2008 Standard for CASE Tool Interconnections—Reference Model for Specifying System Behavior	SW Engineering Models and Methods
IEEE Std. 1220-2005 (a.k.a. ISO/IEC 26702:2007) Standard for Application and Management of the Systems Engineering Process	SW Engineering Process
IEEE Std. 1228-1994 Standard for Software Safety Plans	SW Quality
IEEE Std. 1320.1-1998 Standard for Functional Modeling Language—Syntax and Semantics for IDEF0	SW Engineering Models and Methods
IEEE Std. 1320.2-1998 Standard for Conceptual Modeling Language—Syntax and Semantics for IDEF1X97 (IDEFobject)	SW Engineering Models and Methods
IEEE Std. 1490-2011 Guide—Adoption of the Project Management Institute (PMI®) Standard, A Guide to the Project Management Body of Knowledge (PMBOK® Guide)—Fourth Edition	SW Engineering Management
IEEE Std. 1517-2010 Standard for Information Technology—System and Software Life Cycle Processes—Reuse Processes	SW Engineering Process

Number and Title (listed in order of number)	Most Relevant KA
IEEE Std. 1633-2008 Recommended Practice for Software Reliability	SW Quality
IEEE Std. 12207-2008 (a.k.a. ISO/IEC 12207:2008) Standard for Systems and Software Engineering—Software Life Cycle Processes	SW Engineering Process
IEEE Std. 14102-2010 Standard Adoption of ISO/IEC 14102:2008 Information Technology—Guideline for the Evaluation and Selection of CASE Tools	SW Engineering Models and Methods
ISO/IEC 14143 [six parts] Information Technology—Software Measurement—Functional Size Measurement	SW Requirements
IEEE Std. 14471-2010 Guide—Adoption of ISO/IEC TR 14471:2007 Information Technology—Software Engineering—Guidelines for the Adoption of CASE Tools	SW Engineering Models and Methods
IEEE Std. 14764-2006 (a.k.a. ISO/IEC 14764:2006) Standard for Software Engineering—Software Life Cycle Processes—Maintenance	SW Maintenance
IEEE Std. 15026.1-2011 Trial-Use Standard Adoption of ISO/IEC TR 15026-1:2010 Systems and Software Engineering—Systems and Software Assurance—Part 1: Concepts and Vocabulary	SW Quality
IEEE Std. 15026.2-2011 Standard Adoption of ISO/IEC 15026-2:2011 Systems and Software Engineering—Systems and Software Assurance—Part 2: Assurance Case	SW Quality
ISO/IEC 15026-3 Systems and Software Engineering—Systems and Software Assurance—Part 3: System Integrity Levels	SW Quality
ISO/IEC 15026-4:2012 Systems and Software Engineering—Systems and Software Assurance—Part 4: Assurance in the Life Cycle	SW Quality
IEEE Std. 15288-2008 (a.k.a. ISO/IEC 15288:2008) Standard for Systems and Software Engineering—System Life Cycle Processes	SW Engineering Process
ISO/IEC/IEEE 15289:2011 Systems and Software Engineering—Content of Life-Cycle Information Products (Documentation)	SW Engineering Process
ISO/IEC 15504 [ten parts] Information Technology—Process Assessment	SW Engineering Process
IEEE Std. 15939-2008 Standard Adoption of ISO/IEC 15939:2007 Systems and Software Engineering—Measurement Process	SW Engineering Management
ISO/IEC 15940:2006 Information Technology—Software Engineering Environment Services	SW Engineering Models and Methods
IEEE Std. 16085-2006 (a.k.a. ISO/IEC 16085:2006) Standard for Systems and Software Engineering—Software Life Cycle Processes—Risk Management	SW Engineering Management
ISO/IEC/IEEE 16326:2009 Systems and Software Engineering—Life Cycle Processes—Project Management	SW Engineering Management
ISO/IEC 19501:2005 Information Technology—Open Distributed Processing—Unified Modeling Language (UML) Version 1.4.2	SW Engineering Models and Methods

Number and Title (listed in order of number)	Most Relevant KA
ISO/IEC 19505:2012 [two parts] Information Technology—Object Management Group Unified Modeling Language (OMG UML)	SW Engineering Models and Methods
ISO/IEC 19506:2012 Information Technology—Object Management Group Architecture-Driven Modernization (ADM)—Knowledge Discovery Meta-Model (KDM)	SW Engineering Models and Methods
ISO/IEC 19507:2012 Information Technology—Object Management Group Object Constraint Language (OCL)	SW Engineering Models and Methods
ISO/IEC TR 19759:2005 Software Engineering—Guide to the Software Engineering Body of Knowledge (SWEBOK)	[General]
ISO/IEC 19761:2011 Software Engineering—COSMIC: A Functional Size Measurement Method	SW Requirements
ISO/IEC 20000-1:2011 Information Technology—Service Management—Part 1: Service management system requirements	SW Engineering Process
ISO/IEC 20926:2009 Software and Systems Engineering—Software Measurement—IFPUG Functional Size Measurement Method	SW Requirements
ISO/IEC 20968:2002 Software Engineering—Mk II Function Point Analysis—Counting Practices Manual	SW Requirements
ISO/IEC 24570:2005 Software Engineering—NESMA Functional Size Measurement Method Version 2.1—Definitions and Counting Guidelines for the Application of Function Point Analysis	SW Requirements
IEEE Std. 24748.1-2011 Guide—Adoption of ISO/IEC TR 24748-1:2010 Systems and Software Engineering—Life Cycle Management—Part 1: Guide for Life Cycle Management	SW Engineering Process
IEEE Std. 24748.2-2012 Guide—Adoption of ISO/IEC TR 24748-2:2011 Systems and Software Engineering—Life Cycle Management—Part 2: Guide to the Application of ISO/IEC 15288 (System Life Cycle Processes)	SW Engineering Process
IEEE Std. 24748-3:2012 Guide—Adoption of ISO/IEC TR 24748-3:2011 Systems and Software Engineering—Life Cycle Management—Part 3: Guide to the Application of ISO/IEC 12207 (Software Life Cycle Processes)	SW Engineering Process
ISO/IEC/IEEE 24765:2010 Systems and Software Engineering—Vocabulary	[General]
ISO/IEC TR 24772:2013 Information technology—Programming Languages — Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use	SW Construction
ISO/IEC 24773:2008 Software Engineering—Certification of Software Engineering Professionals	SW Engineering Professional Practice
IEEE Std. 24774:2012 Guide—Adoption of ISO/IEC TR 24474:2010 Systems and Software Engineering—Life Cycle Management—Guidelines for Process Description	SW Engineering Process
ISO/IEC 25000:2005 Software Engineering—Software Product Quality Requirements and Evaluation (SQuaRE)—Guide to SQuaRE	SW Quality

Number and Title (listed in order of number)	Most Relevant KA
ISO/IEC 25000 through 25099 Software Engineering—Software Product Quality Requirements and Evaluation (SQuaRE)	SW Quality
ISO/IEC 25010:2011 Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—System and Software Quality Models	SW Quality
ISO/IEC 25060 through 25064 Software Engineering—Software Product Quality Requirements and Evaluation (SQuaRE)—Common Industry Format (CIF) for Usability	SW Quality
ISO/IEC/IEEE 26511:2012 Systems and Software Engineering—Requirements for Managers of User Documentation	SW Engineering Management
ISO/IEC/IEEE 26512:2011 Systems and Software Engineering—Requirements for Acquirers and Suppliers of User Documentation	SW Engineering Management
IEEE Std. 26513-2010 Standard Adoption of ISO/IEC 26513:2009 Systems and Software Engineering—Requirements for Testers and Reviewers of Documentation	SW Testing
IEEE Std. 26514-2010 Standard Adoption of ISO/IEC 26514:2008 Systems and Software Engineering—Requirements for Designers and Developers of User Documentation	SW Design
ISO/IEC/IEEE 26515:2012 Systems and Software Engineering—Developing User Documentation in an Agile Environment	SW Engineering Models and Methods
ISO/IEC 29110 [several parts] Software Engineering—Lifecycle Profiles for Very Small Entities (VSE)	SW Engineering Process
ISO/IEC/IEEE 29119 [four parts] (Draft) Software and Systems Engineering—Software Testing	SW Testing
ISO/IEC/IEEE 29148:2011 Systems and Software Engineering—Life Cycle Processes—Requirements Engineering	SW Requirements
ISO/IEC/IEEE 42010:2011 Systems and Software Engineering—Architecture Description	SW Design
IEEE Std. 90003:2008 Guide—Adoption of ISO/IEC 90003:2004 Software Engineering—Guidelines for the Application of ISO 9001:2000 to Computer Software	SW Quality

APPENDIX C

CONSOLIDATED REFERENCE LIST

The Consolidated Reference List identifies all recommended reference materials (to the level of section number) that accompany the breakdown of topics within each knowledge area (KA). This Consolidated Reference List is adopted by the software engineering certification and associated professional development products offered by the IEEE Computer Society. KA Editors used the references allocated to their KA by the Consolidated Reference List as their Recommended References.

Collectively this Consolidated Reference List is

- Complete: Covering the entire scope of the *SWEBOK Guide*.
- Sufficient: Providing enough information to describe "generally accepted" knowledge.
- Consistent: Not providing contradictory knowledge nor conflicting practices.
- Credible: Recognized as providing expert treatment.
- Current: Treating the subject in a manner that is commensurate with currently generally accepted knowledge.
- Succinct: As short as possible (both in number of reference items and in total page count) without failing other objectives.

[1*] J.H. Allen et al., *Software Security Engineering: A Guide for Project Managers*, Addison-Wesley, 2008.

[2*] M. Bishop, *Computer Security: Art and Science*, Addison-Wesley, 2002.

[3*] B. Boehm and R. Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*, Addison-Wesley, 2003.

[4*] F. Bott et al., *Professional Issues in Software Engineering*, 3rd ed., Taylor & Francis, 2000.

[5*] J.G. Brookshear, *Computer Science: An Overview*, 10th ed., Addison-Wesley, 2008.

[6*] D. Budgen, *Software Design*, 2nd ed., Addison-Wesley, 2003.

[7*] E.W. Cheney and D.R. Kincaid, *Numerical Mathematics and Computing*, 6th ed., Brooks/Cole, 2007.

[8*] P. Clements et al., *Documenting Software Architectures: Views and Beyond*, 2nd ed., Pearson Education, 2010.

[9*] R.E. Fairley, *Managing and Leading Software Projects*, Wiley-IEEE Computer Society Press, 2009.

[10*] D. Galin, *Software Quality Assurance: From Theory to Implementation*, Pearson Education Limited, 2004.

[11*] E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed., Addison-Wesley Professional, 1994.

[12*] P. Grubb and A.A. Takang, *Software Maintenance: Concepts and Practice*, 2nd ed., World Scientific Publishing, 2003.

[13*] A.M.J. Hass, *Configuration Management Principles and Practices*, 1st ed., Addison-Wesley, 2003.

[14*] E. Horowitz et al., *Computer Algorithms*, 2nd ed., Silicon Press, 2007.

[15*] IEEE CS/ACM Joint Task Force on Software Engineering Ethics and Professional Practices, "Software Engineering Code of Ethics and Professional Practice (Version 5.2)," 1999; www.acm.org/serving/se/code.htm.

[16*] *IEEE Std. 828-2012, Standard for Configuration Management in Systems and Software Engineering*, IEEE, 2012.

[17*] *IEEE Std. 1028-2008, Software Reviews and Audits*, IEEE, 2008.

[18*] *ISO/IEC 14764 IEEE Std. 14764-2006, Software Engineering—Software Life Cycle Processes—Maintenance*, IEEE, 2006.

[19*] S.H. Kan, *Metrics and Models in Software Quality Engineering*, 2nd ed., Addison-Wesley, 2002.

[20*] S. McConnell, *Code Complete*, 2nd ed., Microsoft Press, 2004.

[21*] J. McGarry et al., *Practical Software Measurement: Objective Information for Decision Makers*, Addison-Wesley Professional, 2001.

[22*] S.J. Mellor and M.J. Balcer, *Executable UML: A Foundation for Model-Driven Architecture*, 1st ed., Addison-Wesley, 2002.

[23*] D.C. Montgomery and G.C. Runger, *Applied Statistics and Probability for Engineers*, 4th ed., Wiley, 2007.

[24*] J.W. Moore, *The Road Map to Software Engineering: A Standards-Based Guide*, 1st ed., Wiley-IEEE Computer Society Press, 2006.

[25*] S. Naik and P. Tripathy, *Software Testing and Quality Assurance: Theory and Practice*, Wiley-Spektrum, 2008.

[26*] J. Nielsen, Usability Engineering, 1st ed., Morgan Kaufmann, 1993.

[27*] L. Null and J. Lobur, *The Essentials of Computer Organization and Architecture*, 2nd ed., Jones and Bartlett Publishers, 2006.

[28*] M. Page-Jones, *Fundamentals of Object-Oriented Design in UML*, 1st ed., Addison-Wesley, 1999.

[29*] K. Rosen, Discrete Mathematics and Its Applications, 7th ed., McGraw-Hill, 2011.

[30*] A. Silberschatz, P.B. Galvin, and G. Gagne, *Operating System Concepts*, 8th ed., Wiley, 2008.

[31*] H.M. Sneed, "Offering Software Maintenance as an Offshore Service," *Proc. IEEE Int'l Conf. Software Maintenance* (ICSM 08), IEEE, 2008, pp. 1–5.

[32*] I. Sommerville, *Software Engineering*, 9th ed., Addison-Wesley, 2011.

[33*] S. Tockey, *Return on Software: Maximizing the Return on Your Software Investment*, 1st ed., Addison-Wesley, 2004.

[34*] G. Voland, *Engineering by Design*, 2nd ed., Prentice Hall, 2003.

[35*] K.E. Wiegers, *Software Requirements*, 2nd ed., Microsoft Press, 2003.

[36*] J.M. Wing, "A Specifier's Introduction to Formal Methods," *Computer*, vol. 23, no. 9, 1990, pp. 8, 10–23.