Chapter 1: Software Requirements

Defining a software requirement

In its most basic form, a software requirement is a property that must be exhibited by something to solve a problem in the real world. You can aim to automate part of a task so that someone supports an organization's business processes, corrects existing software deficiencies, or controls a device, to name just a few of the many issues for which software solutions are possible. The ways users work, business processes, and devices are often complex. By extension, therefore, the requirements of a particular software are usually a complex combination of several people at different levels of an organization, and that in one way or another are involved or connected with this feature of the environment in which the software will operate.

Functional requirements describe the functions that the software should execute; for example, formatting a text or modulating a signal. They are sometimes referred to as capabilities or features. A functional requirement can also be described as one for which a finite set of test steps can be written to validate its behavior.

Non-functional *requirements* are those that act to restrict the solution. Non-functional requirements are sometimes referred to as restrictions or quality requirements. They can also be classified according to performance requirements, maintenance requirements, security requirements, reliability requirements, security requirements, interoperability requirements, or one of many other types of software requirements (see Models and quality features in the software quality KA).

System Requirements and Software Requirements In this topic, "system" means an interactive combination of elements to achieve a defined goal. These include hardware, software, firmware, people, information, techniques, installations, services, and other support items. as defined by the International System and Software Engineering Council (INCOSE). System requirements are the system requirements as a whole. On a system that contains software components, software requirements are derived from system requirements. This KA defines "user requirements" in a restricted way, such as the requirements of customers or end users of the system. System requirements, on the other hand, cover user requirements, the requirements of other stakeholders (such as regulatory authorities), and requirements without an identifiable human source.

- Users: This group includes those who will operate the software. It is often a heterogeneous group that involves people with different roles and requirements.
- Customers: This group includes those who have ordered the software or who represent the target software market.
- Market analysts: A mass market product won't have a customer in charge, so it often takes marketing staff to establish what the market needs and act as proxy customers.
- Regulators: Many application domains, such as banking and public transport, are regulated. The software in these domains must meet the requirements of the regulatory authorities.
- Software engineers: These people have a legitimate interest in benefiting from software development, for example by reusing components in other products. If, in this scenario, a customer of a particular product has specific requirements that compromise the reuse potential of components, software engineers must carefully weigh their own involvement against that of customers.

Chapter 2: Software Design

In a general sense, design can be seen as a form of troubleshooting. For example, the concept of a perverse problem—a problem without a definitive solution—is interesting in terms of understanding the limits of design. Several other notions and concepts are also of interest to understand design in its general sense: objectives, constraints, alternatives, representations, and solutions (see Problem solving techniques in Computing Foundations KA). Software design is an important part of the software development process. To understand the role of software design, we need to see how it fits into the software development lifecycle. Therefore, it is important to understand the main features of software requirements analysis, software design, software construction, software testing, and software maintenance.

1. Architectural design (also known as high-level design and top-level design) describes how software is organized into components.
2. The detailed design describes the desired behavior of these components.

Abstraction is "a view of an object that focuses on information relevant to a particular purpose and ignores the rest of the information" (see Abstraction in Computing Foundations KA). In the context of software design, two key abstraction mechanisms are parameterization and specification. Parameterization abstraction abstracts the details of data representations when rendering data as named parameters. Specification abstraction leads to three main types of abstraction: procedural abstraction, data abstraction, and control abstraction (iteration).

Coupling and cohesion. Coupling is defined as "a measure of interdependence between modules in a computer program", while cohesion is defined as "a measure of the force of association of elements within a module" .

Decomposition and modularization. Decomposing and modularizing means that large software is divided into several smaller named components that have well-defined interfaces that describe component interactions. Usually, the goal is to place different functionalities and responsibilities in different components.

Encapsulating and hiding information means grouping and packaging the internal details of an abstraction and making those details inaccessible to external entities.

Interface separation and implementation . Separating the interface and implementation involves defining a component by specifying a public interface (known to clients) that is separate from the details of how the component is performed (see encapsulation and hiding information above).

Sufficiency, integrity and primitivity. Achieving sufficiency and integrity means ensuring that a software component captures all the important features of an abstraction and nothing else. Primitividad means that the design must be based on patterns that are easy to implement.

Separation of concerns. One concern is an "area of interest with respect to software design" [8]. A design concern is a design area that is relevant to one or more of your stakeholders. Each view of the architecture frames one or more concerns. Separating concerns by point of view allows stakeholders to focus on a few things at a time and provides a means to manage complexity

Architectural styles

An architectural style is "a specialization of element types and relationships, along with a set of constraints on how they can beused." Therefore, an architectural style can be considered to provide the high-level organization of the software. Several authors have identified several important architectural styles:

- General structures (e.g. layers, pipes, and filters, whiteboard)
- Distributed systems (for example, client-server, three-tier, broker)
- Interactive systems (e.g. Model-View-Controller, Presentation-Abstraction-Control)
- Adaptive systems (e.g. microkernel,reflection)

- Others (for example, batches, interpreters, process control, rule-based).

3.3 Design patterns

Described succinctly, a pattern is "a common solution to a common problem in a given context"]. While architectural styles can be seen as patterns that describe the high-level organization of the software, other design patterns can be used to describe details at a lower level. These lower-level design patterns include the following:

- Creation patterns (e.g. builder, factory, prototype, singleton)
- Structural patterns (e.g. adapter, bridge, composite, decorator, facade, flyweight,proxy)
- Behavior patterns (for example, command, interpreter, iterator, mediator, memory, observer, status, strategy, template, visitor).

General UI Design Principles

- *Learning capacity.* The software should be easy to learn so that the user can start working quickly with the software.
- *User Familiarity* . The interface should use terms and concepts drawn from the experiences of the people who will use the software.
- *Consistency* . The interface must be consistent for comparable operations to fire in the same way.
- *Minimal surprise.* The behavior of the software should not surprise users.
- *Recoverability* . The interface must provide mechanisms that allow users to recover from errors.
- *User orientation* . The interface should provide meaningful feedback when errors occur and provide context-related help to users.
- *Diversity of users* . The interface should provide appropriate interaction mechanisms for various types of users and for users with different capabilities (blind, visually impaired, deaf, color blind, etc.)

Chapter 3: Building Software

1.1 Minimizing complexity

Most people have a limited ability to maintain complex structures and information in their working memories, especially for long periods of time. This proves to be an important factor influencing how

people transmit intent to computers and leads to one of the strongest drivers in software construction: minimizing complexity. The need to reduce complexity applies essentially to all aspects of software construction and is particularly critical for testing software builds. In software construction, reduced complexity is achieved by emphasizing code creation that is simple and readable rather than intelligent. It is achieved through the use of standards, modular design and many other specific techniques. It is also backed by construction-focused quality techniques.

1.2 Anticipating change

Most software will change over time and anticipation of change drives many aspects of software construction; Changes in the environments in which the software operates also affect the software in a variety of ways. Anticipating change helps software engineers create extensible software, which means they can improve a software product without disrupting the underlying structure. Anticipation of change is supported by many specific techniques

1.3 Construction for verification

Building for verification means building software in such a way that software engineers who write the software can easily find faults, as well as testers and users during independent testing and operational activities. Specific techniques that support construction for verification include following coding standards to support code reviews and unit tests, organizing code to support automated testing, and restricting the use of complex or difficult-to-understand language structures, among others.

1.4 Reuse

Reuse refers to the use of existing assets to solve different problems. In software construction, typical assets that are reused include libraries, modules, components, source code, and out-of-the-box commercial assets (COTS). Reuse is best practiced systematically, according to a repeatable and well-defined process. Systematic reuse can enable significant improvements in software productivity, quality, and costs. Reuse has two closely related facets: "construction for reuse" and "construction with reuse". The first means creating reusable software assets, while the second means reusing software assets in building a new solution. Reuse often transcends project boundaries, meaning that the assets used can be built in other projects or organizations.

1.5 Building standards

The application of internal or external development standards during construction helps achieve the efficiency, quality and cost objectives of a project. Specifically, the options for subsets of allowed programming languages and usage standards are important aids for greater security. Standards that directly affect construction problems include

- communication methods (e.g. standards for document formats and content)
- programming languages (for example, language standards for languages such as Java and C++) * coding standards (for example, standards for naming, design, and indentation conventions)
- platforms (for example, interface standards for operating system calls)
- tools (for example, diagrammering standards for notations such as UML (Unified Modeling Language)).

Use of external standards. Construction depends on the use of external standards for building languages, construction tools, technical interfaces, and interactions between Software Construction KA and other KA. Standards come from numerous sources, including hardware and software interface specifications (such as Object Management Group (OMG)) and international organizations (such as IEEE or ISO). Use of internal standards. Standards can also be created on an organizational basis at the corporate level or for use in specific projects. These standards support the coordination of the group's activities, minimizing complexity, anticipating change and building for verification

CHAPTER 4. SOFTWARE TESTS

It is an activity that allows to evaluate and improve the quality of the product in order to detect faults and correct errors.

Software testing consists of dynamically verifying the behavior of a program through a finite group of duly selected test cases. The perception that software testing has been changed only at the end of the source code creation process, and it is very useful to do so at all stages of software development; this allows you to correct errors and detect background failures, on time.

SOFTWARE TEST BASICS

- Test-related terminology
- Key elements
- Relationship of evidence with other activities

TEST LEVELS

- The object of the test
- Test objectives

TEST TECHNIQUES

- Evidence based on intuition and experience
- Specification-based techniques
- Code-based techniques
- Error-based techniques
- Use-based techniques
- Techniques based on the nature of the application
- Selecting and combining techniques

TEST MEASURES

- Evaluating a program through testing
- Evaluation of tests carried out

TESTING PROCESS

- Practical considerations
- Testing activities

CHAPTER 5. SOFTWARE MAINTENANCE

The software development process must meet the requirements posed, once the process of covering defects, operation and change of environment must take place at this stage, the maintenance phase begins with a period of warranty and post-implementation support,but the maintenance of the software occurs much earlier.

Although the maintenance stage of the software has not had the degree of attention that this type of software development is already starting to change since many serious errors have occurred because it does not pay attention to it that it deserves.

Software maintenance has been defined as the total number of activities required to provide effective software support, including effective planning before. During and after software deployment.

Fundamentals in software maintenance:

The IEEE/EIA 12207 standard defines maintenance as one of the main processes in the software lifecycle, the goal is to modify the existing software while preserving its integrity so does ISO/IEC 14764 this emphasizes previous deliveries for software maintenance planning.

The need for maintenance is given to ensure that the software satisfactorily meets the requested requirements, it applies to any development independent of the lifecycle model used, maintenance is given in order to achieve the proper performance and in the order of:

- Fix faults.
- Improvise the design.
- Implement fixes.
- Interfaces with other systems.

A number of key factors we need to keep in mind to ensure effective software maintenance, it is important to understand that software maintenance provides us with a unique technique in management challenges for software engineers, we can appreciate how subsequent releases are planned as well as patches generated for previous versions, which follows presents us with a way to how we are presented with some management and technical factors for software maintenance, these are grouped according to the following topics:

- Technical factors.
- Administrative factors.
- Cost estimation.

CHAPTER 6. SOFTWARE CONFIGURATION MANAGEMENT

Configuration management is the discipline responsible for identifying the overall configuration of a system to maintain its reliability, adaptability, and configuration to different lifecycles. It is formally defined by IEEE610.12-90 as "Discipline applied technically and administratively for management and survival to: Identify and document the physical and functional characteristics of the configuration of the elements, control in the change of their characteristics record and report changes in the implementation process, as well as their status and verification of compliance with their specific requirements".

1. Management of SCM processes

The SCM manages and controls the evolution and integrity of the software, as well as its verification, control, reports and configuration of information. Successful implementation of SCM requires special care and planning and management.

2. Identification of software settings

It identifies the elements to be controlled, establishes and identifies schemas and their versions, establishes tools and techniques used to manage and control those elements.

3. Software configuration control

You are concerned about lifecycle change management, covering the processes that determine the changes to be made, the authority to make them, and support for the implementation of those changes.

The information derived from these activities is useful for measuring change traffic and breaking down aspects to be redoed.

4. Recording the status of the Software configuration

Software Configuration Status Accounting (SCSA) is the activity of recording and providing the information needed for effective software configuration management.

5. Auditing software configuration

It is an independently developed activity to assess the conformity of software products, is responsible for applying regulations, standards, guidance plans and procedures.

CHAPTER 7. MANAGEMENT OF SOFTWARE ENGINEERING

It can be defined as application management activities, planning, coordination, measurement, monitoring, control and reports to ensure the development and maintenance of the software as systematic, disciplined and quantifiable.

Aspects of organization management are important in terms of the impact on software engineering and management policies, these policies can be influenced by the requirements of effective software, maintenance, and development.

1. Initiation and Scope

It focuses on the effective determination of software requirements through various induction methods and the assessment of project feasibility from different points of view, once feasibility is established, the to-do is to specify the validation of requirements and the change of procedures.

2. Planning a Software Project

It is regulated by scopes and requirements and by the feasibility of the project, life cycle processes are evaluated and the most appropriate one is selected.

3. Enactment of the Software Project

The plans are implemented and the processes included in the plans are disclosed, in this process there is total expectation of full adherence to the contractor's requirements and the achievement of the project objectives, are fundamental activities for the promulgation of project management, measurement, supervision, control and information.

4. Review and Evaluation

The overall process towards achieving the objectives and satisfaction of contractor requirements is evaluated and assessments are carried out on the effectiveness of the overall process to date, the personnel involved and the tools and methods used.

5. Close

The project comes to an end when all the plans and processes involved have been enacted and completed, at this stage certain criteria for the success of the project are reviewed.

6. Software engineering measures

Here we address the issue of measurement in software engineering and its importance for this follows metrics and standards established by entities such as ISO and IEEE.

Chapter 8: Software Engineering Process.

Software development tools are computer-aided instruments that are required to assist software lifecycle processes. The instruments allow repeated actions, well defined to be automated, reducing the cognitive burden on the software engineer who is then free to concentrate on the creative aspects of the process.

Instruments often designed to support particular software engineering methods, reducing any administrative burden associated with applying the method by hand. Like software engineering methods, they are required to make the software that plots more systematic, vary in the scope of supporting individual tasks that embrace in the entire lifecycle.

Software engineering methods impose the structure on software engineering activity in order to make the activity systematic and ultimately more likely to be successful. Methods typically provide notation and vocabulary, procedures for performing identifiable tasks, and guidelines for checking both the process and the product. They vary widely in scope, from a single phase of the lifecycle to the entire lifecycle. The emphasis on this area of knowledge is on software engineering methods spanning multiple phases of the lifecycle, as phase-specific methods are covered by other areas of knowledge.

While there are detailed manuals on specific instruments and numerous research papers on innovative instruments, technical generic writings on software engineering instruments are relatively scarce. One difficulty is high rate of change of software instruments in general. Specific details change regularly, making it difficult to provide concrete examples to update them.

Chapter 9: Software Engineering Models and Methods.

The KA of the software engineering process can be examined at two levels. The first level encompasses the technical and management activities within the software lifecycle processes carried out during the acquisition, development, maintenance and removal of the software.

The second is a meta-level, which refers to the definition, implementation, valuation, measurement, management, changes and improvements of the software lifecycle processes themselves. The first level is covered by the other KAs in the Guide. This KA takes care of the second level.

The term "software engineering process" can be interpreted in a variety of ways, and this can lead to confusion:

- Standards like IEEE talk about software engineering processes, which means there are many processes involved, such as development processes or management configuration process.
- A second meaning refers to a general discussion of processes related to software engineering. this is the meaning that is intended with the title of this KA and the one most commonly used in the description of KA.

- Finally, a third meaning could refer to the current set of activities performed within an organization, which could be seen as a single process, especially from within the organization, this meaning is used in the KA in very few cases.

Software engineering processes matter not only to large organizations, but process-related activities can, and have been, successfully performed by small organizations, teams, and individuals.

Chapter 10: Software Quality.

Software engineering management can be defined as the application for management activities – planning, coordination, measurements, monitoring, control and reporting that ensures the development and maintenance of systematic, disciplined and quantified software.

KA Software Engineering Management is therefore responsible for the management and measurement of software engineering. Although measuring is an important aspect in all KA's it is not so far that the topic of measurement programs is presented.

Although it is true, on the one hand, to state that, in a sense, it should be possible to manage software engineering in the same way that any other process exists specific to software products and software lifecycle processes that complicate effective management—only some of which are listed below:

- Customer perception is such that there is often a lack of appreciation for the complexity inherent in software engineering, particularly in relation to the impact of changing requirements.

- It is almost inevitable that the software engineering processes themselves will generate the need for new or changed customer requirements.

- Software engineering necessarily incorporates aspects of creativity and discipline – maintaining an appropriate balance between the two is often difficult.

- The degree of novelty and complexity of the software are often extremely high.

- The exchange rate of the underlying technology is very fast.

With regard to software engineering, management activities take place at three levels: organizational and infrastructure management, project management, and planning and measurement control program.

Chapter 11: Professional Software Engineering Practice.

To circumscribe software engineering, it is necessary to identify the disciplines with which software engineering shares a common limit. This chapter identified, in alphabetical order, these related disciplines. For her role, related disciplines also share several common boundaries between themselves.

Computing Curricula 2001 project (CC2001) draft computer engineering volume *reportstates* that *"Computer*engineering incorporates the science and technology of thedesign, construction, implementation and maintenance of the software and hardware components of modern computing systems and computer-controlled equipment. *"*

This report identifies the following areas of knowledge (known as areas in the report) for computer engineering:

- Algorithms and complexity
- Computer architecture and organization
- Computer systems engineering
- Circuits and systems
- Digital logic
- Direct structures
- Digital signal processing
- Direct structures
- Programming fundamentals
- Algorithms and complexity
- Architecture and organization
- Operating systems

- Centralized systems
- Programming languages
- Man-Machine Interaction
- Graphics and visual computing
- Smart systems
- Information management
- Social and professional problems
- Software engineering
- Computational science and numerical calculation

Chapter 12: Software Engineering Economics

The economics of software engineering are about making decisions related to software engineering in an enterprise context. The success of a software product, service, and solution depends on good business management. However, in many companies and organizations, commercial software relationships with software development and engineering remain vague. This area of knowledge (KA) provides an overview of the economics of software engineering. Economics is the study of value, costs, resources and their relationship in a particular context or situation. In the discipline of software engineering, activities have costs, but the resulting software itself has economic attributes as well. The software engineering economy provides a way to study software attributes and software processes in a systematic way that relates them to economic measures. These economic measures can be weighed and analyzed by making decisions that fall within the scope of a software organization and those within the integrated scope of an entire production or acquisition business. The software engineering economy aligns technical software decisions with the organization's business objectives. In all types of organizations, whether "for profit", "non-profit" or government, this results in a sustainable stay in the business. In "for-profit" organizations, this also relates to achieving tangible return on invested capital, both assets and capital spent. This KA has been formulated in a way that addresses all types of organizations regardless of approach, product and service portfolio, or capital ownership and tax restrictions. Decisions like "Should we use a specific component?" It may seem easy from a technical perspective, but it can have serious implications for the commercial viability of a software project and the resulting product. Engineers often wonder if such concerns apply at all, as they are "engineers only". Economic analysis and decision-making are important engineering considerations because engineers are able to evaluate decisions both technically and from a business perspective. The contents of this area

of knowledge are important topics that software engineers should know, even if they are never involved in specific business decisions; They will have a complete view of business problems and the role technical considerations play in business decision-making. Many engineering proposals and decisions, such as dealing with buying, have profound intrinsic economic impacts that must be explicitly considered. This KA first covers the fundamentals, key terminology, basic concepts, and common practices of the software engineering economy to indicate how decision-making in software engineering includes, or should include, a business perspective. It then provides a lifecycle perspective, highlights risk and uncertainty management, and shows how economic analysis methods are used. Some practical considerations end the area of knowledge. highlights risk and uncertainty management, and shows how economic analysis methods are used. Some practical considerations end the area of knowledge. highlights risk and uncertainty management, and shows how economic analysis methods are used. Some practical considerations end the area of knowledge.

Chapter 13: Fundamentals of Computing

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

In general, it is accepted that software engineering is based on computing. For example, "Software Engineering 2004: Curriculum Guidelines for Software Engineering Bachelor's Programs" clearly states: "A particularly important aspect is that software engineering is based on computing and mathematics" (emphasis added).

Steve Tockey wrote in his book Return on Software: Both computer science and software engineering deal with computers, computing and software. Computer science, as a body of knowledge, is at the core of both. Software engineering is responsible for the application of computers, computing and software for practical purposes, specifically the design, construction and operation of efficient and economical software systems. Therefore, at the heart of software engineering is the understanding of computing.

While few people will deny the role of computing in the development of software engineering as a discipline and as a body of knowledge, the importance of

computing for software engineering cannot be overstated; therefore, this Ka of Computing Fundamentals is being written.

Most of the topics discussed in Computing Foundations KA are also topics of discussion in basic courses taught in undergraduate and graduate programs in computer science. Such courses include programming, data structure, algorithms, computer organization, operating systems, compilers, databases, networks, distributed systems, etc. Therefore, when breaking down topics, it can be tempting to break down the Computing Fundamentals KA according to these divisions that are often found in relevant courses.

However, a purely course-based topic division suffers from serious drawbacks. On the one hand, not all computer courses are related or equally important to software engineering. Therefore, some topics that would otherwise be covered in a computer course are not covered in this KA. For example, computer graphics, although it is an important course in a bachelor's degree program in computer science, is not included in this KA.

Second, some topics discussed in this guide do not exist as separate courses in undergraduate or graduate computer science programs. As a result, these topics may not be adequately covered in a purely course-based breakdown. For example, abstraction is a topic incorporated into several different computer courses; It is unclear which course abstraction should belong to in a course-based topic breakdown.

The Computing Foundations KA is divided into seventeen different themes. The direct utility of a topic for software engineers is the criterion used to select the topics to include in this KA. The advantage of this topic-based breakdown is that it is based on the belief that Computing Foundations,if you want to understand firmly, should be considered as a collection oflogically connected topics that underpin software engineering in general and software construction in particular.

The Computer Fundamentals KA is closely related to software design, software construction, software testing, software maintenance, software quality and mathematical foundations.

Chapter 14: Mathematical Fundamentals

Software professionals live with the programs. In a very simple language, one can program only for something that follows a well-understood and unapoguous logic. The Mathematical Fundamentals (KA) area of knowledge helps software engineers understand this logic, which in turn translates into programming language code. The mathematics that are the main focus in this KA are quite different from the typical

arithmetic, where numbers are discussed and discussed. Logic and reasoning are the essence of the mathematics that a software engineer must address.

Mathematics, in a sense, is the study of formal systems. The word "formal" is associated with precision, so there cannot be an ambiguous or erroneous interpretation of the fact. Therefore, mathematics is the study of each and every truth certain about any concept. This concept can be both numbers and symbols, images, sounds, videos, almost anything. In short, not only numbers and numerical equations are subject to precision. In contrast, a software engineer needs to have accurate abstraction in a different application domain.

The SWEBOK Ka Mathematical Fundamentals guide covers basic techniques for identifying a set of rules for reasoning in the context of the system under study. All that can be inferred by these rules is absolute certainty within the context of that system. In this KA, techniques that can represent and advance the reasoning and judgment of a software engineer in an accurate (and therefore mathematical) manner are defined and discussed. The language and methods of logic discussed here allow us to describe mathematical tests to conclusively infer the absolute truth of certain concepts beyond numbers. In short, you can write a program for a problem only if it follows some logic. The goal of this KA is to help you develop the ability to identify and describe this logic.

Chapter 15: Engineering Fundamentals

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

To fulfill their responsibilities, engineers must understand how different product and process characteristics vary. Engineers often encounter situations in which it is necessary to study the relationship between different variables. An important point to keep in mind is that most studies are conducted on the basis of samples, so the observed results should be understood with respect to the entire population. Therefore, engineers must develop an adequate understanding of statistical techniques to collect reliable data in terms of sampling and analysis to reach results that can be generalized. These techniques are discussed below.

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

Measurements can be physical, environmental, economic, operational or some other type of measurement that is significant for the particular project. This section explores measurement theory and how it is fundamental to engineering.

IEEE defines engineering as "the application of a systematic, disciplined and quantifiable approach to structures, machines, products, systems or processes". This chapter describes some of the fundamental engineering skills and techniques that are useful to a software engineer. The focus is on topics that support other KA While minimizing duplication of topics covered elsewhere in this document.

As the theory and practice of software engineering matures, it is increasingly clear that software engineering is an engineering discipline that is based on knowledge and skills common to all engineering disciplines. This Engineering Fundamentals (KA) area of knowledge deals with the engineering fundamentals that apply to software engineering and other engineering disciplines. The topics of this KA include empirical methods and experimental techniques; statistical analysis; measurement; engineering design; modeling, prototyping and simulation; standards; and root cause analysis. Applying this knowledge, as appropriate, will enable software engineers to develop and maintain the software more efficiently and effectively. Completing your engineering work efficiently and effectively is a goal of all engineers in all engineering disciplines.