SWEBOK: Software Requirements Engineering Knowledge Area Description

Version 0.5

Pete Sawyer and Gerald Kotonya
Computing Department,
Lancaster University
United Kingdom
{sawyer} {gerald} @comp.lancs.ac.uk

Table of contents

1. Introduction

This document proposes a breakdown of the SWEBOK Software Requirements Engineering Knowledge Area. The knowledge area was originally proposed as 'Software Requirements Analysis'. However, as a term to denote the whole process of acquiring and handling of software requirements, 'Requirements Analysis' has been largely superceded by 'Requirements Engineering'. We therefore use 'Requirements Engineering' to denote the knowledge area and 'Requirements Analysis' as one of the activities that comprise SRE.

Requirements engineering is of great economic importance to the software industry. The cost of fixing errors in the system requirements tends to increase exponentially the longer they remain undetected. Projects waste enormous resources rewriting code that enshrines mistaken assumptions about customers' needs and environmental constraints. There is therefore considerable risk involved in requirements engineering yet the maturity of most software development organizations' requirements engineering processes lags well behind that of their down-stream life-cycle processes. This is compounded by the fact that requirements engineering is tightly time- and resource-bounded and time spent analysing requirements is often wrongly perceived to be unproductive time. To help move forward from this situation, this knowledge area description sets out a proposal for the fundamental knowledge needed by software engineers to perform effective requirements engineering.

The issue of who should perform requirements engineering is, however, controversial. Requirements engineering requires a set of skills that intersect with those of software engineering but which also includes skills outside of this set. It is also important Requirements engineering is not always a 'software' only activity. For certain classes of system (e.g. embedded systems), it may need expert input from disciplines. For this reason, we refer in this document to the 'requirements engineer' rather than the 'software engineer'. This is not to suggest that software engineers cannot 'do' requirements engineering, but rather to emphasize that the role of the requirements engineer is a distinct one that mediates between the domain of software development and the domain of the software's customer(s) (application domain). Requirements engineering may not be the preserve of software engineers but it should be performed by people with software engineering knowledge.

2. References used

All the references from the jump-start documents are used here, namely [Dor97, Pfl98, Pres97, Som96, Tha97]. [Dor97, Pfl98, Pres97, Som96] place requirements engineering in the context of software engineering. This is a useful first step in understanding requirements engineering. [Tha97] is an extensive collection of papers tackling many issues in requirements engineering, and most of the topics that make up the knowledge area (section .

This initial list of references has been supplemented with additional requirements engineering specific and other relevant references[1]. Key additional references are [Dav93, Kot98, Lou95, Som97, Tha90, Tha93]. [Dav93] contains extensive discussion on requirements structuring, models and analysis. [Kot98] is an extensive text on requirements engineering covering most of the knowledge area topics. [Som97] is a useful practitioner's guide to good requirements engineering. [Tha90] is an extensive and useful collection of standards and guidelines for requirements engineering.

3. System requirements

At its most basic, a computer-based system requirement can be understood as a property that the system must exhibit in order for it to adequately perform its function. This function may be to automate some part of a task of the people who will use the system, to support the business processes of the organisation that has commissioned the system, controlling a device in which the software is to be embedded, and many more. The functioning of the users, or the business processes or the device will typically be complex and, by extension, the requirements on the system will be a complex combination of requirements from different people at different levels of an organisation and from the environment in which the system must execute.

[1] [Ama97, And96, Che90,Gog93, Hal96, Har93, Hum88, Hum89, Pau96, Sid96, Rud94]

Clearly, requirements will vary in intent and in the kinds of properties they represent. A crude distinction is often drawn between:

- Capabilities that the system must provide, such as saving a file or modulating a signal. Capabilities are sometimes known as functional requirements.
- Constraints that the system must operate under, such as having a probability of generating a fatal error during any hour of operation of less than $1 * 10^{-8}$. Constraints are sometimes known as non-functional or quality requirements.
- A constraint on the development of the system (e.g. 'the system must be developed using Ada').
- A specific constraint on the system (e.g. 'the sensor must be polled 10 times per second').

In practice, this distinction is hard to apply rigidly and it is difficult to formulate a consistent definition of the distinction between them. Constraints are particularly hard to define and tend to vary from vaguely expressed goals to very specific bounds on the software's performance. Examples of these might be: that the software must increase the call-center's throughput by 20%; and that the task will return a result within 10ms. In the former case, the requirement is at a very high level and will need to be elaborated into a number of specific capabilities which, if all satisfied by the software, should achieve the required improvement in throughput. In the latter case, the requirement will constrain the manner in which the software engineers implement the functionality needed to compute the result.

This also illustrates the range of levels at which requirements are formulated. The standard, though rather unsatisfactory, way of distinguishing levels of requirements is as 'user' or 'system' requirements. User requirements are the direct requirements of the people who use or otherwise have a stake in (the 'stakeholders') the system. They consider the system to be a 'black box' and are only concerned with what is externally exhibited by the system. Their collection, organisation and analysis is crucial to gaining a clear understanding of the problem for which the computer system is to provide a solution and its likely cost. This understanding has to be validated and trade-offs negotiated before further resources are committed to the project. The user requirements therefore need to be expressed in terms that allow the stakeholders to do this validation. This usually means the requirements are expressed using a structured form of natural language to describe the system's effect on the stakeholders' business domain. Exceptionally, this may be supplemented by models using specialized software and systems engineering notations, but used carefully so that they can be understood with minimal explanation.

Software engineers who are experts in the stakeholders' domain are very rare so they cannot be expected to take a list of user requirements, interpret their meaning and translate them into a configuration of software and other system components that satisfy the user requirements. In most cases, therefore, user requirements are elaborated by the requirements engineer (together with *expert* input on the application domain), into a number of more detailed requirements that more precisely describe what the system must do. This usually entails deploying engineering skills to construct models of the system in order to understanding the logical partitioning of the system, its context in the operational environment and the data and control communications between the logical entities. A side-effect of this is that new requirements (emergent properties) will emerge as a conceptual architecture of the system starts to take shape. It is also likely to reveal problems with the user requirements (such as missing information) which have to be resolved in partnership with the stakeholders.

The result of this process is a set of detailed and more rigorously defined requirements. These are the system requirements (software and system requirements). Because the intended readership is technical, the system requirements can make use of modeling notations to supplement their textual descriptions. The requirements enable a system architect to propose a solution architecture. They form the baseline for the real development work so they should enable detailed planning and costs to be derived.

The above description implies a deterministic process where user requirements are elicited from the stakeholders, elaborated into system requirements and passed over to the development team. This is an idealized view. In practice, many things conspire to make requirements engineering one of the riskiest and most poorly understood parts of the software life cycle. The requirements engineer may interpret the stakeholders' requirements wrongly, the stakeholders may have difficulty articulating their requirements, or

technology, market conditions or the customer's business processes may change mid-way through development.

All of these things militate against the ideal of having a requirements baseline frozen and in place before development begins. Requirements *will* change and this change must be managed by continuing to 'do' requirements engineering throughout the life-cycle of the project. In a typical project the activities of the requirements engineer evolve over time from elicitation and modeling to impact analysis and trade-off negotiation. To do this effectively requires that the requirements are carefully documented and managed so that the impact of proposed changes can be traced back to the affected user requirements and forwards to the affected implementation components.

What we have described above refers to requirements on the system (the 'product'). Requirements can also be imposed on how the system is developed (the 'process'). Process requirements are also important because they can impact on development schedules, development costs and the skills required of the developers. Thus, it is important to (for example) understand the implications of working for a customer who requires that the software contractor has achieved accreditation to CMM level 3. An important process requirement that every project has is a cost constraint.

4. The context of software requirements engineering

For many types of system it is impossible to separate the requirements for the software from the broader requirements for the system as a whole. As well as software, the system may include computer hardware and other types of hardware device which are interfaced to the computer and operational processes which are used when the system is installed in some working environment.

Computer-based systems fall into two broad types:

1. User-configured systems where a purchaser puts together a system from existing software products. The vast majority of personal computer systems are of this type. ‘Systems engineering’ is the responsibility of the buyer of the software, which is installed on a general-purpose computer. The software requirements for this type of system are created by the companies which develop the different software products from customer requests and from their perception of what is marketable.

2. Custom or bespoke systems where a customer produces a set of requirements for a hardware/software system and a contractor develops and delivers that system. The customer and the contractor may be different divisions within the same organization or may be separate companies. The system requirements describe services to be provided by the system as a whole and, as part of the systems engineering process, the specific requirements for the software in the system are derived. Custom systems vary in size from very small embedded systems in consumer devices such as microwaves ovens, to gigantic command and control systems such as military messaging systems.

   There are three important classes of custom systems:
   - Information systems
     These are systems that are primarily concerned with processing information that is held in some kind of database. They are usually implemented using standard computer hardware (e.g. mainframe computers, workstations, PCs) and are built on top of commercial operating systems. For these systems, requirements engineering is primarily software requirements engineering.
   - Embedded systems
     These are systems where software is used as a controller in some broader hardware system, They range from simple systems (e.g. in a CD player) to very complex control systems (e.g. in a chemical plant). They often rely on special purpose hardware and operating systems. Requirements engineering for these systems involves both hardware and software requirements engineering.
   - Command and control systems
     These are, essentially, a combination of information systems and embedded systems where special-purpose computers provide information, which is collected and stored in a database and then used to help people make decision. These systems usually involve a mix of different types of

computer, which are networked in some way. Within the whole system, there may be several embedded systems and information systems. Air traffic control systems, railway signaling systems, military communication systems are examples of command and control systems. Requirements engineering for these systems includes hardware and software specification and a specification of the operational procedures and processes.

The complete requirements engineering process spans several activities from the initial statement of requirements through to the more detailed development specific software requirements. The process takes place after a decision has been made to acquire a system. Before that, there are activities concerned with establishing the business need for the system, assessing the feasibility of developing the system, and setting the budget for the system development.

The requirements engineering process often includes some initial design activity where the overall structure (architecture) of the system is defined. This activity has to be carried out at this stage to help structure the system requirements and to allow the requirements for different sub-systems to be developed in parallel.

The requirements engineering processes will vary across organizations and projects. Important factors that are likely to influence this variation include:

- The nature of the project. A market-driven project that is developing a software product for general sale imposes different demands on the requirements engineering process than does a customer-driven project that is developing bespoke software for a single customer. For example, a strategy of incremental releases is often used in market-driven projects while this is often unacceptable for bespoke software. Incremental release imposes a need for rigorous prioritisation and resource estimation for the requirements to select the best subset of requirements for each release. Similarly, requirements elicitation is usually easier where there is a single customer than where there are either only potential customers (for a new product) or thousands of existing but heterogeneous existing customers (for a new release of an existing product).
- The nature of the application. Software requirements engineering means different things to (for example) information systems and embedded software. In the former case, a software development organisation is usually the lead or the only contractor. The scale and complexity of information systems projects can cause enormous difficulties, but at least the software developer is involved from initial concept right through to hand-over. For embedded software the software developer may be a subcontractor responsible for a single subsystem and have no direct involvement with eliciting the user requirements, partitioning the requirements to subsystems or defining acceptance tests. The requirements will simply be allocated to the software subsystem by a process that is opaque to the software contractor. A complex customer / main contractor / subcontractor project hierarchy is just one of many factors that can greatly complicate resolution of any problems that emerge as the software contractor analyses the allocated requirements.

Because of the artificiality in distinguishing between system and software requirements, we henceforth omit the word ‘software’ and talk about ‘requirements engineering’ except where it is helpful to make a distinction. This is also reflected by the coverage of requirements engineering by standards. Requirements documentation is the only aspect of requirements engineering that is covered by dedicated standards. Wider requirements engineering issues tend to be covered only as activities of software engineering or systems engineering. Current process improvement and quality standards offer only limited coverage of requirements engineering issues despite its place at the root of software quality problems.

5. Software requirements engineering breakdown: processes and activities

There are broadly two common ways of decomposing requirements engineering knowledge. The first is exemplified by the European Space Agency's software engineering standard PSS-05 [Maz96]. Here, the knowledge area is decomposed at the first level into the 2 topics of user requirements definition and software requirements specification. This is broadly what we described in section 2 and is a useful starting point (which is why we described it). However, as a basis for the knowledge area breakdown it confuses process activities and products with knowledge and practice.

Process knowledge is crucial but we prefer to separate out process issues from knowledge. The knowledge area topics correspond approximately to process tasks but they tend to be enacted concurrently and

iteratively rather than in series as is implied by a breakdown based on PSS-05's model. The breakdown we have chosen broadly corresponds to that in [Som97 and Kot98] and is orthogonal to PSS-05. It comprises 5 topics:

- The requirements engineering process.
- Requirements elicitation.
- Requirements analysis.
- Requirements validation.
- Requirements management.

Figure 1 shows the coarse-grain requirements engineering process model.

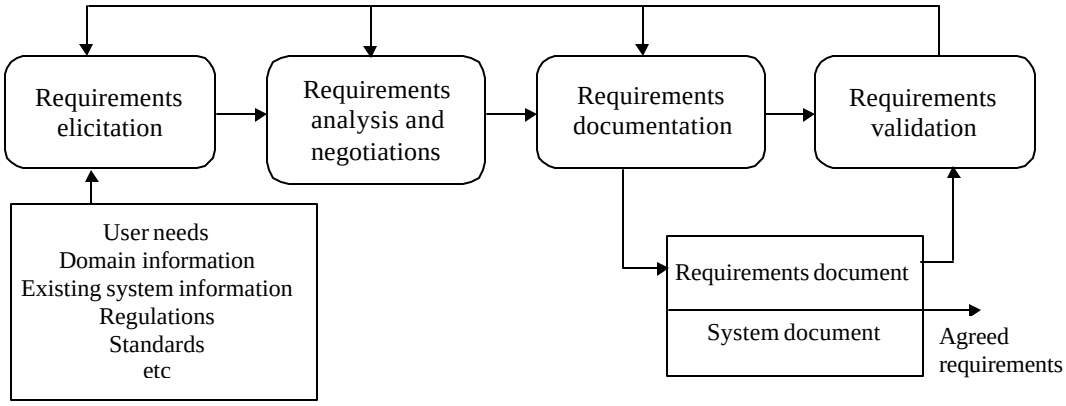

Figure 1 Coarse grain requirements engineering process

5.1 The requirements engineering process

This section is concerned with introducing the requirements engineering process and to orient the remaining 4 topics and show how requirements engineering dovetails with the overall software life-cycle. This section also deals with contractual and project organisation issues. The topic is broken down into 4 subtopics.

5.1.1 Process models.

This subtopic is concerned with introducing a small number of generic process models. The purpose is to lead to an understanding that the requirements process:

- is not a discrete front-end activity of the software life-cycle but rather a process that is initiated at the beginning of a project but continues to operate throughout the life-cycle;
- is connected to the overall software process by the life-cycle of requirements and the involvement of the requirements engineer;
- will need to be tailored to the organisation and project context.

In particular, the subtopic provides the context that allows the software engineer to understand how requirements elicitation, analysis, validation and management fit together.

5.1.2 Process actors.

This subtopic introduces the roles of the people who enact the requirements engineering process. The key people are the system stakeholders (users and other people who have a stake in the software) and the requirements engineer. The software engineering roles of (for example) system architect, tester and quality assurance are also described with reference to requirements engineering.

5.1.3 Process support.

This subtopic introduces the resources required and consumed by the requirements engineering process. This includes human resources, training requirements, methods and tools.

5.1.4 Process improvement.

This subtopic is concerned with quality. Its purpose is to emphasize the key role requirements engineering plays in terms of the cost, timeliness and customer satisfaction of software products. It will help orient the requirements engineering process with quality standards and process improvement models for software and systems.

Links to common themes	
Quality	Software process improvement (SPI) recognizes the link between process and software quality. The process improvement subtopic is the one most concerned with quality here. This contains links to SPI standards such as the software and systems engineering CMMs, the forthcoming ISO/IEC 15504 (SPICE) and ISO 9001-3. Requirements engineering is at best peripheral to these and only work to address requirements engineering processes specifically, is the requirements engineering good practice guide (REGPG) [Som 97].
Standards	SPI standards as above. In addition, life-cycle software engineering standards ISO/IEC 12207 and ESA PSS-05 apply, in part, to the software requirements engineering process.
Measurement	Measurement is poorly developed for software requirements engineering processes. Where measurement is applied at all it is at a coarse level of granularity: e.g. total numbers of requirements, numbers of requirements changes.
Tools	We are aware of no application of tools to manage the requirements engineering process although there may be a role for workflow.

5.2 Requirements elicitation

This topic covers what is sometimes termed 'requirements capture', 'requirements discovery' or 'requirements acquisition'. It is concerned with where requirements come from and how they can be collected by the requirements engineer. Requirements elicitation is the first stage in building an understanding of the problem the software is required to solve. It is fundamentally a human activity and is where the stakeholders are identified and relationships established between the development team (usually in the form of the requirements engineer) and the customer. There are 2 main subtopics.

5.2.1 Requirements sources

In a typical system, there will be many sources of requirements and it is essential that all potential sources are identified and evaluated for their impact on the system. This subtopic is designed to promote awareness of different requirements sources and frameworks for managing them. The main points covered are:

- Goals. The term 'Goal' (sometimes called 'business concern' or 'critical success factor') refers to the overall, high-level objectives of the system. Goals provide the motivation for a system but are often vaguely formulated. Requirements engineers need to pay particular attention to assessing the impact and feasibility of the goals.
- Domain knowledge. The requirements engineer needs to acquire or to have available knowledge about the application domain. This enables them to infer tacit knowledge that the stakeholders don't articulate, inform the trade-offs that will be necessary between conflicting requirements and sometimes to act as a 'user' champion.
- System stakeholders. Many systems have proven unsatisfactory because they have stressed the requirements for one group of stakeholders at the expense of others. Hence, systems are delivered that are hard to use or which subvert the cultural or political structures of the customer organisation. This subtopic is designed to alert the requirements engineer to the need to identify, represent and manage the 'viewpoints' of many different types of stakeholder.
- The operational environment. Requirements will be derived from the environment in which the software will execute. These may be, for example, timing constraints in a real-time system or

interoperability constraints in an office environment. These must be actively sought because they can greatly affect system feasibility and cost.

- The organizational environment. Many systems are required to support a business process and this may be conditioned by the structure, culture and internal politics of the organisation. The requirements engineer needs to be sensitive to these since, in general, new software systems should not force unplanned change to the business process.

5.2.2 Elicitation techniques

When the requirements sources have been identified the requirements engineer can start eliciting requirements from them. This subtopic concentrates on techniques for getting human stakeholders to articulate their requirements. This is a very difficult area and the requirements engineer needs to be sensitized to the fact that (for example) users may have difficulty describing their tasks, may leave important information unstated, or may be unwilling or unable to cooperate. It is particularly important to understand that elicitation is not a passive activity and that even if cooperative and articulate stakeholders are available, the requirements engineer has to work hard to elicit the right information. A number of techniques will be covered but the principal ones are:

- Interviews. Interviews are a 'traditional' means of eliciting requirements. It is important to understand the advantages and limitations of interviews and how they should be conducted.
- Observation. The importance of systems' context within the organizational environment has led to the adaptation of observational techniques for requirements elicitation whereby the requirements engineer learns about users' tasks by immersing themselves in the environment and observing how users interact with their systems and each other. These techniques are relatively new and expensive but are instructive because they illustrate that many user tasks and business processes are too subtle and complex for their actors to describe easily.
- Scenarios. Scenarios are valuable for providing context to the elicitation of users' requirements. They allow the requirements engineer to provide a framework for questions about users' tasks by permitting 'what if?' and 'how is this done?' questions to be asked. There is a link to 5.3.2. (conceptual modeling) because recent modeling notations have attempted to integrate scenario notations with object-oriented analysis techniques.
- Prototypes. Prototypes are a valuable tool for clarifying unclear requirements. They can act in a similar way to scenarios by providing a context within which users better understand what information they need to provide. There are a range of prototyping techniques which range from paper mock-ups of screen designs to beta-test versions of software products. There is a strong overlap with the use of prototypes for requirements validation (5.4.2).

Links to common themes	
Quality	The quality of requirements elicitation has a direct effect on product quality. The critical issues are to recognise the relevant sources, to strive to avoid missing important requirements and to accurately report the requirements.
Standards	Only very general guidance is available for elicitation from current standards.
Measurement	N/A
Tools	Elicitation is relatively poorly supported by tools. Some modern modeling tools support notations for scenarios. Several programming environments support prototyping but the applicability of these will depend on the application domain. A number of tools are becoming available that support the use of viewpoint analysis to manage requirements elicitation. These have had little impact to date.

5.3 Requirements analysis

This subtopic is concerned with the process of analysing requirements to:

- detect and resolve conflicts between requirements;
- discover the bounds of the system and how it must interact with its environment;

- elaborate user requirements to software requirements.

The traditional view of requirements analysis was to reduce it to conceptual modeling using one of a number of analysis methods such as SADT or OOA. While conceptual modeling is important, we include the classification of requirements to help inform trade-offs between requirements (requirements classification), and the process of establishing these trade-offs (requirements negotiation).

5.3.1 Requirements classification

Requirements can be classified on a number of dimensions. Common useful classifications include:

- As capabilities or constraints. Distinguishing between capabilities and constraints can be hard but is worth-while, particularly where the software's 'qualities' (in safety-related system, for example) are especially important. It is common to sub-classify constraints as, for example, performance, security, reliability, maintainability, etc. requirements. Many such requirements imply the need for special metrics to specify and validate the requirements and have implications for the resources that need to be allocated to testing. It is useful to perform this classification early in the requirements engineering process to help understand the full implication of user requirements.
- According to priority. In general, it will be impossible to implement every user requirement because of mutual incompatibilities between requirements and because of insufficient resources. These conflicting demands have to be traded-off and one of the most important items of information needed to do this is the requirements' priorities. In practice, it can be hard to do this and for many organizations a 3-level classification such as mandatory/highly desirable/desirable provides a crude but workable scheme.
- According to the requirements' cost/impact. This classification complements that of priority since if the cost of implementing a mandatory requirement is unaffordable, this will reveal a serious problem with the stakeholder's expectations. The impact of changing a requirement is also important since it strongly affects the cost of doing so.
- According to their scope. The scope of a requirement can also affect its priority and cost. Requirements with global scope (for example goals - see 5.2.1) tend to be costly and of high priority; responsibility for their satisfaction cannot be allocated to a single component of the architecture.
- According to volatility/stability. Some requirements will change during the life-cycle of the software and even during the development process itself. It is sometimes useful if some estimate of the likelihood of a requirement changing can be made. For example, in a banking application, requirements for functions to calculate and credit interest to customers' accounts are likely to be more stable than a requirement to support a particular kind of tax-free account. The former reflect a fundamental feature of the banking domain (that accounts can earn interest), while the latter may be rendered obsolete by a change to government legislation. Flagging requirements that may be volatile can help the software engineer establish a design that is more tolerant of change.
- According to whether they are product or process requirements.

Other classifications may be appropriate, depending upon the development organization's normal practice and the application itself. There is a strong overlap between requirements classification and requirements attributes (5.5.1).

5.3.2 Conceptual modeling

The development of models of the problem is fundamental to requirements analysis. The purpose is to aid understanding of the problem rather than to initiate design of the solution. However, the boundary between requirements engineering and design is often a blurred one in practice and the requirements engineer may find themselves unavoidably having to model aspects of the solution. There are several kinds of models that can be developed. These include data and control flows, state models, event traces, user interactions, object models and many others. The factors that influence the choice of model include:

- The nature of the problem. Some types of application demand that certain aspects be analysed particularly rigorously. For example, control flow and state models are likely to be more important for real-time systems than for an information system.
- The expertise of the requirements engineer. It is generally better to adopt a modeling notation or method that the requirements engineer has experience with.

- The process requirements of the customer. Customers may impose a particular notation or method on the requirements engineer. This can conflict with the last factor.
- The availability of methods and tools. Notations or methods that are poorly supported by training and tools may not reach widespread acceptance even if they are suited to particular types of problem.

Note that in almost all cases, it is useful to start by building a model of the 'system boundary'. This is crucial to understanding the system's context in its operational environment and identify its interfaces to the environment.

The issue of modeling is tightly coupled with that of methods. For practical purposes, a method is a notation (or set of notations) supported by a process that guides the application of the notations. Methods and notations come and go in fashion. Object-oriented notations are currently in vogue (especially UML) but the issue of what is the 'best' notation is seldom clear. There is little empirical evidence to support claims for the superiority of one notation over another.

Formal modeling using notations based upon discrete mathematics and which are tractable to logical reasoning have made an impact in some specialized domains. These may be imposed by customers or standards or may offer compelling advantages to the analysis of certain critical functions or components.

This topic does not seek to 'teach' a particular modeling style or notation but rather to provide guidance on the purpose and intent of modeling.

5.3.3 Requirements negotiation

Another name commonly used for this subtopic is 'conflict resolution'. It is concerned with resolving problems with requirements where conflicts occur; between two stakeholders' requiring mutually incompatible features, or between requirements and resources or between capabilities and constraints, for example. In most cases, it is unwise for the requirements to make a unilateral decision so it is necessary to consult with the stakeholder(s) to reach a consensus on an appropriate trade-off. It is often important for contractual reasons that such decisions are traceable back to the customer. We have classified this as a requirements analysis topic because problems emerge as the result of analysis. However, a strong case can also be made for counting it as part of requirements validation.

Links to common themes	
Quality	The quality of the analysis directly affects product quality. In principle, the more rigorous the analysis, the more confidence can be attached to the software quality.
Standards	Software engineering standards stress the need for analysis. More detailed guidance is provided only by de-facto modeling 'standards' such as UML.
Measurement	Part of the purpose of analysis is to quantify required properties. This is particularly important for constraints such as reliability or safety requirements where suitable metrics need to be identified to allow the requirements to be quantified and verified.
Tools	There are many tools that support conceptual modeling and a number of tools that support formal specification. There are a small number of tools that support conflict identification and requirements negotiation through the use of methods such as quality function deployment.

5.4 Requirements validation

It is normal for there to be one or more formally scheduled points in the requirements engineering process where the requirements are validated. The aim is to pick up any problems before resources are committed to addressing the requirements.

One of the key functions of requirements documents is the validation of their contents. Validation is concerned with checking the documents for omissions, conflicts and ambiguities and for ensuring that the requirements follow prescribed quality standards. The requirements should be necessary and sufficient and should be described in a way that leaves as little room as possible for misinterpretation. There are four important subtopics.

5.4.1 The conduct of requirements reviews.
Perhaps the most common means of validation is by the use of formal reviews of the requirements document(s). A group of reviewers is constituted with a brief to look for errors, mistaken assumptions, lack of clarity and deviation from standard practice. The composition of the group that conducts the review is important (at least one representative of the customer should be included for a customer-driven project, for example) and it may help to provide guidance on what to look for in the form of checklists.

Reviews may be constituted on completion of the user requirements definition document, the software requirements specification document, the baseline specification for a new release, etc.

5.4.2 Prototyping.
Prototyping is commonly employed for validating the requirements engineer's interpretation of the user requirements, as well as for eliciting new requirements. As with elicitation, there is a range of prototyping techniques and a number of points in the process when prototype validation may be appropriate. The advantage of prototypes is that they can make it easier to interpret the requirements engineer's assumptions and give useful feedback on why they are wrong. For example, the dynamic behaviour of a user interface can be better understood through an animated prototype than through textual description or graphical models. There are also disadvantages, however. These include the danger of users attention being distracted from the core underlying functionality by cosmetic issues or quality problems with the prototype. They may also be costly to develop although if they avoid the wastage of resources caused by trying to satisfy erroneous requirements, their cost can be more easily justified.

5.4.3 Model validation.
The quality of the models developed during analysis should be validated. For example, in object models, it is useful to perform a static analysis to verify that communication paths exist between objects that, in the stakeholders domain, exchange data. If formal specification notations are used, it is possible to use formal reasoning to prove properties of the specification (e.g. completeness).

5.4.4 Acceptance tests.
An essential property of a user requirement is that it should be possible to verify that the finished product satisfies the requirement. Requirements that can't be verified are really just 'wishes'. An important task is therefore planning how to verify each requirement. In most cases, this is done by designing acceptance tests. One of the most important requirements quality attributes to be checked by requirements validation is the existence of adequate acceptance tests.

Links to common themes	
Quality	Validation is all about quality - both the quality of the requirements and of the documentation.
Standards	Software engineering life-cycle and documentation standards (e.g. IEEE std 830) exist and are widely used in some domains to inform validation exercises.
Measurement	Measurement is important for acceptance tests and definitions of how requirements are to be verified.
Tools	Some limited tool support is available for model validation and theorem provers can assist developing proofs for formal models.

5.5 Requirements management
Requirements management is an activity that should span the whole software life-cycle. It is fundamentally about change management and the maintenance of the requirements in a state that accurately mirrors the software to be, or that has been, built.

There are 4 main topics concerned with requirements management.

5.5.1 Change management
Change management is central to the management of requirements. This subtopic is intended to describe the role of change management, the procedures that need to be in place and the analysis that should be applied to proposed changes. It will have strong links to the configuration management knowledge area.

5.5.2 Requirements attributes
Requirements should consist not only of a specification of what is required, but also of ancillary information that helps manage and interpret the requirements. This should include the various classifications attached to the requirement (see 4.3.1) and the verification method or acceptance test plan. It may also include additional information such as a summary rationale for each requirement, the source of each requirement and a change history. The most fundamental requirements attribute, however, is an identifier that allows the requirements to be uniquely and unambiguously identified. A naming scheme for generating these IDs is an essential feature of a quality system for a requirements engineering process.

5.5.3 Requirements tracing
Requirements tracing is concerned with recovering the source of requirements and predicting the effects of requirements. Tracing is fundamental to performing impact analysis when requirements change. A requirement should be traceable backwards to the requirements and stakeholders that motivated it (from a software requirement back to the user requirement(s) that it helps satisfy, for example). Conversely, a requirement should be traceable forwards into requirements and design entities that satisfy it (for example, from a user requirement into the software requirements that have been elaborated from it and on into the code modules that implement it).

The requirements trace for a typical project will form a complex directed acyclic graph (DAG) of requirements. In the past, development organizations either had to write bespoke tools or manage it manually. This made tracing a short-term overhead on a project and vulnerable to expediency when resources were short. In most cases, this resulted in it either not being done at all or being performed poorly. The availability of modern requirements management tools has improved this situation and the importance of tracing (and requirements management in general) is starting to make an impact in software quality.

5.5.4 Requirements documentation
This subtopic is concerned with the role and readership of requirements documentation, with documentation standards and with the establishment of good practice.

Requirements documents are the normal medium for recording and communicating requirements. It is good practice to record the user requirements and the software requirements in different documents. The user requirements definition document organizes the user requirements and makes them available to the stakeholders for validation. Once validated, the user requirements can be elaborated and recorded in the software requirements specification document (SRS). Once this has been validated, it serves as the basis for subsequent development.

Requirements documents need to be subjected to version control as requirements change. Modern requirements management tools allow requirements to be documented in electronic form along with their associated attributes and traceability links. Such tools usually include document generators that allow requirements documents to be generated in defined formats so, for example, the software requirements can be used to (semi-) automatically generate an SRS that conforms to IEEE std 830.

Links to common themes	
Quality	Requirements management is a level 2 key practice area in the software CMM and this has boosted recognition of its importance for quality.
Standards	Software engineering life-cycle and documentation standards (e.g.

	IEEE std 830) exist and are widely used in some domains.
Measurement	Mature organizations may measure the number of requirements changes and use quantitative measures of impact assessment.
Tools	There are a number of requirements management tools on the market such as DOORS and RTM.

Appendix A - Summary of requirements engineering breakdown

Requirements engineering topics	**Subtopics**
The requirement engineering process	Process models Process actors Process support Process improvement
Requirements elicitation	Requirements sources Elicitation techniques
Requirement analysis	Requirements classification Conceptual modelling Requirements negotiation
Requirements validation	Requirements reviews Prototyping Model validation Acceptance tests
Requirements management	Change management Requirements attributes Requirements tracing Requirements documentation

Appendix B Matrix of topics and references

In Table 1 shows the topic/reference matrix. The table is organized according to requirements engineering topics in Appendix A. A 'X' indicates that the topic is covered to a reasonable degree in the reference. A 'X' in appearing in main topic but not the sub-topic indicates that the main topic is reasonably covered (in general) but the sub-topic is not covered to any appreciable depth. This situation is quite common in most software engineering texts, where the subject of requirements engineering is viewed in the large context of software engineering.

TOPIC	[Dav93]	[Dor97]	[Kot98]	[Lou95]	[Maz96]	[Pfl98]	[Pre97]	[Som96]	[Som97]	[Tha90]	[Tha97]
Requirements engineering process	X	X	X	X				X	X	X	X
Requirement process models			X	X				X	X	X	X
Requirement process actors	X		X	X					X	X	X
Requirement process support									X	X	X
Requirement process improvement			X						X		
Requirements elicitation	X	X	X	X		X			X	X	X
Requirements sources	X		X	X		X	X		X	X	X
Elicitation techniques	X		X	X		X				X	X
Requirements analysis	X	X	X	X			X	X	X	X	X
Requirements classification	X	X	X	X				X	X		X
Conceptual modeling	X		X	X			X	X		X	X
Requirements negotiation			X						X		
Requirements validation	X	X		X				X	X	X	X
Requirements reviews			X	X					X		X
Prototyping	X		X	X					X	X	X
Model validation			X						X		X
Acceptance tests	X								X	X	
Requirements management	X	X	X	X				X	X	X	X
Change management			X						X		X
Requirement attributes			X						X		X
Requirements tracing			X						X	X	X
Requirements documentation	X		X		X		X	X	X		X

Table 1 Topics and their references

Appendix C - Software requirements engineering and Bloom's taxonomy

Appendix D – Software requirements engineering references

[Ama97] K, EL Amam, J. Drouin, et al. *SPICE: The theory and Practice of Software Process Improvement and Capability Determination*. IEEE Computer Society Press, 1997.

[And96] S.J. Andriole, *Managing Systems Requirements: Methods, Tools and Cases*. McGraw-Hill, 1996.

[Che90] P. Checkland and J. Scholes, *Soft Sysems Methodology in Action.* John Wiley and Sons, 1990.

[Dav93] A.M. Davis, *Software Requirements: Objects, Functions and States*. Prentice-Hall, 1993.

[Dor97] M. Dorfman and R.H. Thayer, *Software Engineering*. IEEE Computer Society Press, 1997.

[Gog93] J.A. Goguen and C. Linde, Techniques for requirements elicitation, Proc. Intl' Symp. On Requirements Engineering. IEEE Press, 1993.

[Hal96] A. Hall, Using Formal Methods to Develop an ATC Information System. IEEE Software 13(2), pp66-76, 1996.

[Har93] R. Harwell. et al, What is a Requirement. Proc 3rd Ann. Int'l Symp. Nat'l Council Systems Eng., pp17-24, 1993

[Hum89] W. Humphery, *Managing the Software Process*. Addison Wesley, 1989.

[Hum88] W. Humphery, Characterizing the Software Process, IEEE Software 5(2): 73-79, 1988.

[Pfl98] S.L. Pfleeger, *Software Engineering-Theory and Practice*. Prentice-Hall, 1998.

[Pres97] R.S. Pressman, *Software Engineering: A Practitioner's Approach (4 edition).* McGraw-Hill, 1997.

[Tha90] R.H. Thayer and M. Dorfman, *Standards, Guidelines and Examples on System and Software Requirements Engineering*. IEEE Computer Society, 1990.

[Tha97] R.H. Thayer and M. Dorfman, *Software Requirements Engineering (2nd Ed)*. IEEE Computer Society Press, 1997.

[Kot98] G. Kotonya, and I. Sommerville, *Requirements Engineering: Processes and Techniques*. John Wiley and Sons, 1998.

[Lou95] P. Loucopolos and V. Karakostas, *System Requirements Engineering.* McGraw-Hill, 1995.

[Maz96] C. Mazza, J. Fairclough, B. Melton, et al, *Software Engineering Guides*. Prentice-Hall, 1996.

[Pau96] M.C. Paulk, C.V. Weber, et al., *Capability Maturity Model: Guidelines for Improving the Software Process.* Addison-Wesley, 1995.

[Sid96] J. Siddiqi and M.C. Shekaran, Requirements Engineering: The Emerging Wisdom, IEEE Software, pp15-19, 1996.

[Som96] I. Sommerville, *Software Engineering (5th edition).* Addison-Wesley, 1996.

[Som97] I. Sommerville and P. Sawyer, *Requirements engineering: A Good Practice Guide.* John Wiley and Sons, 1997

[Rud94] J. Rudd and S. Isense, Twenty-two Tips for a Happier, Healthier Prototype. ACM Interactions, 1(1), 1994.