SWEBOK: Software Requirements Engineering Knowledge Area Description

Version 0.6

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

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a term for the systematic acquisition and handling of software requirements, 'requirements analysis' has been largely superceded by 'requirements engineering'.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' to denote an activity of requirements engineering. Requirements engineering has been defined as follows [Som97]:

Requirements engineering is a relatively new term which has been invented to cover all of the activities involved in discovering, documenting, and maintaining a set of requirements for a computer-based system. The use of the term 'engineering' implies that systematic and repeatable techniques should be used to ensure that system requirements are complete, consistent, relevant, etc.

It is significant that this definition excludes the prefix 'software' from requirements engineering and talks instead about system requirements. In order to derive the requirements on the software, it is necessary to identify the requirements on the (computer-based, software-intensive or socio-technical) system of which the software will be a part. Ultimately, software requirements must satisfy real world needs and hence we cannot sensibly consider software requirements without first understanding the system requirements. For this reason, requirements engineering is fundamentally an activity of systems engineering rather than one that is specific to software engineering. In the remainder of this document we drop the 'Software' prefix and refer to 'Requirements Engineering'.

Given that requirements engineering is not specific to software, it is reasonable to ask why it should be a distinct knowledge area of the software engineering body of knowledge. The answer is that good requirements engineering is crucial to the software industry. The cost of fixing errors in the system requirements tends to increase exponentially the longer they remain undetected. Projects routinely waste enormous resources rewriting code that enshrines mistaken assumptions about customers' needs and environmental constraints. Similarly, customers are frequently asked to accept software that fails to deliver the expected functionality or quality and which forces unplanned changes to their business processes. There is therefore considerable risk involved in requirements engineering yet few software development organizations make best use of the measures that are available to mitigate these risks. 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.

The requirements engineer's job is a hence a difficult one. They must:

- collate imperfect information from a range of disparate sources;
- use this information to build an understanding of complex problem domains and their constraints;
- use this understanding to recommend and negotiate appropriate trade-offs;
- distil the information into a requirements specification that provides a sound foundation for system development;
- and manage the requirements so that any enforced changes are controlled and their effects contained.

To do all this they need to mediate between the domain of the system users and customers, and the technical domain of the systems and software engineer.

In this document we use the term 'requirements engineer' as a synonym for individuals or teams who perform the job of requirements engineering. In some organisations this is a specialised role. In many others it is just one of the roles of software or systems engineers. This knowledge area description sets out a proposal for the fundamental knowledge needed by software engineers to perform effective requirements engineering.

We have tried to avoid domain dependency in the document. The knowledge area document is really about identifying requirements engineering practice *and* identifying when the practice is and isn't appropriate. We do recognise that desktop software products are different from reactor control systems and the document should be read in this light. Where we refer to particular tools, methods, notations, SPI models, etc. it does not imply our endorsement of them. They are merely used as examples.

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.

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. Overview of Requirements Engineering

This section provides an overview of requirements engineering in which we define the notion of a 'requirement'; justify our assertion that requirements engineering is concerned with systems rather than software engineering; discuss motivations for systems and consider how these relate to system requirements; describe a generic process for analysis of requirements and follow this with a discussion of why, in practice, organisations often deviate from this process; and describe the deliverables of the requirements engineering process and the need to manage requirements. This overview is intended to provide a perspective or 'viewpoint' on the knowledge area that complements the one in section 4 - the knowledge area breakdown.

Readers who are familiar with requirements engineering concepts and terms are invited to skip to section 4.

3.1. What is a requirement?

At its most basic, a requirement can be understood as a property that a 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, to control 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. By extension, therefore, 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.

Requirements vary in intent and in the kinds of properties they represent. A distinction can be drawn between *product parameters* and *process parameters*. Product parameters are requirements on the system to be developed and can be further classified as:

- Functional requirements on the system such as formatting some text or modulating a signal. Functional requirements are sometimes known as capabilities.
- Non-functional requirements that act to constrain the solution. Non-functional requirements are sometimes known as constraints or quality requirements. They can be further classified according to whether they are (for example) performance requirements, maintainability requirements, safety requirements, reliability requirements, electro-magnetic compatibility requirements and many other types of requirements.

A process parameter is essentially a constraint on the development of the system (e.g. 'the software must be written in Ada'). These are sometimes known as process requirements.

Non-functional requirements are particularly hard to handle and tend to vary from vaguely expressed goals to specific bounds on how the software must behave. Two examples of these might be: that the system must increase the call-center's throughput by 20%; and a reliability requirement that the system shall have a probability of generating a fatal error during any hour of operation of less than $1 * 10^{-8}$. The throughput requirement is at a very high level and will need to be elaborated into a number of specific functional requirements. The reliability requirement will tightly constrain the system architecture.

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

An essential property of all requirements is that they should be verifiable. Unfortunately, non-functional requirements may be difficult to verify. For example, it is impossible to design a test that will demonstrate that the above reliability requirement has been satisfied. Instead, it will be necessary to construct simulations and perform statistical tests from which the system's probable reliability can be inferred. This will be very costly and illustrates the need to define non-functional requirements that are appropriate to the application domain yet not so stringent as to be beyond the bounds of the project budget.

Non-functional requirements should be quantified. If a non-functional requirement is only expressed qualitatively, it should be further analysed until it is possible to express it quantitatively. Non-functional requirements should never be expressed so vaguely as to be unverifiable ('the system shall be reliable', 'the user interface shall be user-friendly').

Stringent non-functional requirements often generate implicit process requirements. The choice of verification method is one example. Another might be the use of particularly rigorous analysis techniques (such as formal specification methods) to reduce systemic errors that can lead to inadequate reliability.

In a typical project there will be a large number of requirements derived from different sources and expressed at different levels of detail. In order to permit these to be referenced and managed, it is essential that each be assigned a unique identifier. There are a number of possible naming or numbering schemes that can be selected that reflect (for example) requirements' derivations or the architectural component to which they are allocated. Whichever scheme is adopted, it is essential that each requirement is unambiguously and uniquely referenced by its identifier.

[Har93] contains a good discussion of the various dimensions against which a requirement may be classified.

3.2. The context of requirements engineering

In the introduction, we drew a distinction between system and software requirements and asserted that requirements engineering is an activity of systems engineering. For many people, systems engineering is about the process of developing custom-built hardware artefacts that contain embedded software – aircraft, railway signalling systems, etc. For these people, it implies a development life-cycle designed to cope with the problems of long life-cycles and of integrating different technologies and component suppliers. However, the notion of systems engineering we use here is inclusive of the development of all types of software-intensive or socio-technical system. For example, many modern systems are built as configurations of off-the-shelf software components such as object request brokers, database management systems and cryptographic modules. There may be no hardware involved in these systems' designs, and there may be relatively little custom software developed to integrate the components. However, the process of developing such systems is fundamentally one of systems engineering. This is because the use of off-the-shelf components, and the capabilities of the system's human users, constrain the way in which the requirements can be satisfied. This is essentially the same as the way in which the properties of aluminium, the capabilities of sensors and actuators, and the physical phenomena of airflow and ice accumulation constrain the way in which the requirements on an aircraft's control systems are addressed.

Only a subset of the requirements for most systems are implemented in software. This is obviously true of systems that comprise electronic, electro-mechanical and software components. However, it is also true of any socio-technical system where tasks are shared between software and humans. A business system, for example, is simply a system for doing work. A modern business system will be supported by software, but it will invariably include people too. The software will support, and be situated in, the business process. It is the business process that is the 'system' that the requirements engineer needs to understand in order to derive the requirements for the software. The requirements engineer's job cannot be restricted to analysing only the requirements to be implemented in software because this assumes *a-priori* knowledge of the overall system requirements and how the system should be partitioned. It is the requirements engineer's job to discover the system requirements, determine the scope of the system and recommend a partitioning of requirements between people, software and other technologies that makes best use of the capabilities of each.

3.3. System requirements and process drivers

The literature on requirements engineering sometimes calls system requirements user requirements. We prefer a restricted definition of the term 'user requirements' in which they denote the requirements of the people who will be the system customers or end-users. System requirements, by contrast, are inclusive of user requirements, requirements of other stakeholders (such as regulatory authorities) and requirements that do not have an identifiable human source. Typical examples of system stakeholders include (but are not restricted to):

- Users – the people who will operate the system. Users are often a heterogeneous group comprising people with different roles and requirements.
- Customers – the people who have commissioned the system or who represent the system's target market.
- Market analysts – a new mass-market product will not have any existing customers so marketing people are often employed to establish what the market needs and to act as proxy customers.
- Regulators – many application domains such as banking and public transport are regulated. Systems in these domains must comply with the requirements of the regulatory authorities.
- System developers – these have a legitimate interest in profiting from developing the system. A common requirement is that costs be shared across product lines so one customer's requirements may be in conflict with the developer's wish to sell the product to other customers. For a mass market product, the developer will be the primary stakeholder since they wish to maintain the product in as large a market as possible for as long as possible.

In addition to these human sources of requirements, important system requirements often derive from other devices or systems in the environment which require some services of the system or act to constrain the system, or even from fundamental characteristics of the application domain. For example, a business system may be required to inter-operate with a legacy database, many military systems have to be tolerant of high levels of electro-magnetic radiation, and mass-market software products often derive requirements from analysis of competing products. We talk of 'eliciting' requirements but in practice the requirements engineer discovers the requirements from a combination of human stakeholders, the system's environment, feasibility studies, market studies, business plans, and domain knowledge.

In this document we use the term 'system requirements' to mean the requirements (functional, non-functional and process) that express a need to address some problem in a particular business context. This problem may take many forms. Common examples include the need to adapt to new business conditions, or to exploit a new business opportunity offered by new markets. Hence, the system may be required to automate a manufacturing process, to support a business task or to exploit a new generation of graphics hardware technology in the computer games market. The point is that this problem is an artefact of the problem context and not of the technology – software or otherwise – with which it may be addressed. The problem is not always technology-neutral, of course, because some systems are expressly intended to exploit a business opportunity offered by a new technology. However, it is a common problem for development projects to be technology-driven, paying too little regard for the real needs of the system customers. It is important therefore, that premature decisions about the nature of the solution are not made until the problem is sufficiently well understood to permit informed decisions to be made. Reflecting this, Alan Davis [Dav93] characterises requirements engineering as comprising two principal activities: problem analysis followed by product description.

It is all too common for systems to be misconceived because their root motivation is never uncovered. In such cases, the system requirements are derived from an erroneous understanding of the system goals. Unless identified early, this misunderstanding propagates through the life-cycle causing expensive revisions, or worse, much later on. For this reason, it is often useful to make an effort to uncover the real motivating business needs for the system. With reference to the examples above, these might be to reduce manufacturing costs, to reduce payroll costs, or simply a make-or-break attempt to stay in business.

Only if the business goals are understood, can we be confident that a new or modified system offers the best solution. Again, with reference to the same three examples, it might turn out that the best solutions are to relocate the manufacturing operation to somewhere with lower labour costs, to take measures to reduce the incidence of staff sickness, or to quit the computer games business. This means that before substantial resources are committed, a business case for the system should be made. This is not really the job of the requirements engineer. However, before commencing detailed work, the requirements engineer should ensure that the customer has thought through their real business needs. Once the business needs are understood, a feasibility study should be conducted to check that they are feasible.

3.4. Requirements analysis in outline

Once the goals of the project have been established, the work of eliciting, analysing and validating the system requirements can commence. This is crucial to gaining a clear understanding of the problem for which the system is to provide a solution and its likely cost.

The system requirements have to be validated by the stakeholders and trade-offs negotiated before further resources are committed to the project. To enable validation, the system requirements are normally kept at a high level and expressed in terms of the domain of the system's customer rather than in technical terms. Hence the system requirements for an Internet book store will be expressed in terms of books, authors, warehousing and credit card transactions, not in terms of the communication protocols, or key distribution algorithms that may form part of the software solution. Too much technical detail at this stage obscures the essential characteristics of the system viewed from the perspective of its customer and users.

Not all of the system requirements will be realisable. Some may be technically infeasible, others may be too costly to implement and some will be mutually incompatible. The requirements engineer must analyse the requirements to understand their implications and how they interact. They must be prioritised and their costs estimated. The goal is to identify the scope of the system and a ‘baseline’ set of system requirements that is feasible and acceptable. This may necessitate helping stakeholders whose requirements conflict (with each other or with cost or other constraints) to negotiate acceptable trade-offs.

To help the analysis of the system requirements, conceptual models of the system are constructed. These aid understanding of the logical partitioning of the system, its context in the operational environment and the data and control communications between the logical entities. It is often helpful to develop prototypes to help explore the users’ requirements.

The system requirements must be analysed in the context of all the applicable constraints. Constraints come from many sources, such as the business environment, the customer’s organisational structure and the system’s operational environment. They include cost, technical (non-functional requirements), regulatory and other constraints. Hence, the requirements engineer’s job is not restricted to eliciting stakeholders’ requirements, but includes identifying the reasons why their requirements may be unrealisable.

It is also crucial that no important requirements are overlooked. The requirements engineer must ensure that all the relevant sources of requirements are identified and consulted. Of course, it will be infeasible to consult everyone. There may be many of users of a large system, for example. However, representative examples of each class of system stakeholder should be identified and consulted. Although individual stakeholders will be authoritative about aspects of the system that represent their interests or expertise, the requirements engineer will be the only one with the ‘big picture’ and so the assurance of completeness rests entirely with them.

Unnecessary requirements should be excluded. The requirements engineer must avoid the common temptation of both users and developers to ‘gold plate’ systems. This is a waste of resources and benefits no-one, least of all the customers, in the long run. The essential principle is that the requirements should be *necessary and sufficient* – there should be nothing left out or anything that doesn’t need to be included.

The requirements engineer must also establish how implementation of the system requirements will be verified. The means of verification must be established and acceptance tests derived that will assure compliance with the requirements before delivery or release of the product.

It is not usually feasible for software developers to begin designing a solution from the baselined system requirements. This is because engineers with expert knowledge of the customer's domain, and so able to accurately interpret the system requirements, are rare. The more complex and specialised the domain, the more of a problem this is. It is therefore necessary to interpret the system requirements for the software engineer.

To do this, the requirements engineer elaborates the system requirements in an iterative process of analysis and refinement. From the system requirements, a number of more detailed requirements that more precisely describe the system are derived. It is at this point that the requirements engineer’s job overlaps with that of the system architect. In many cases, these are two roles played by the same person. A system architecture is derived that partitions the system into subsystems and components to which responsibility for the satisfaction of subsets of the requirements are allocated. The system architecture may be derived form the conceptual models of the system. However, a conceptual model represents the problem and its business context, while the system architecture is a model of the solution. Hence, there may not be a one-to-one mapping between conceptual model and solution architecture components.

Derivation of the system architecture represents a major milestone in the project and it is crucial to get the architecture right because once defined, and resources are committed, the architecture is hard to change.

As the architecture of the system starts to take shape, new requirements (emergent properties) will emerge. Interface requirements between system components are a typical example. Errors and quality problems with the system requirements are also likely to be revealed at this stage and will have to be resolved by again consulting the stakeholders, seeking domain expertise, building prototypes or other means. The requirements allocated to software are the software requirements. These may need to undergo a further cycle of analysis and elaboration. They form the baseline for the real software development work and should enable detailed plans and costs to be derived.

3.5. Requirements engineering in practice

While the general aims of the analysis process described above is fairly generic, it will not be appropriate in every case. There is often insufficient time, effort or freedom from implementation constraints to permit an orderly process such as that described in section 3.4. There is a general pressure in the software industry for ever shorter development cycles, and this is particularly pronounced in highly competitive market-driven sectors. Moreover, relatively few projects are 'green field'. Most are constrained in some way by their environment and many are upgrades to or revisions of existing systems where the system architecture is a given. In practice, therefore, it is almost always impractical to implement requirements engineering as a linear, deterministic process where system requirements are elicited from the stakeholders, elaborated into software requirements, baselined and passed over to the software development team. It is certainly a myth that the requirements are ever perfectly understood or perfectly specified.

Instead, requirements typically iterate toward a level of quality and detail that is *sufficient* to permit design and procurement decisions to me made. In some projects, this may result in the requirements being baselined before all their properties are fully understood. This is not desirable, but it is often a fact of life in the face of tight time pressure.

Even where more resources are allocated to requirements engineering, the level of analysis will seldom be uniformly applied. For example, early on in the process experienced engineers are often able to identify where existing or off-the-shelf solutions can be adapted to the implementation of system components. The requirements allocated to these need not be elaborated further, while others, for which a solution is less obvious, may need to be subjected to further analysis. Critical requirements, such as those concerned with safety, must be analysed especially rigorously.

In almost all cases requirements understanding evolves in parallel with design and development, often leading to the revision of requirements late in the life-cycle. This is perhaps the most crucial point of understanding about requirements engineering - a significant proportion of the requirements *will* change. Often, as much as 40% [Hut95] of the system requirements will require some degree of rework during the product life-time because of errors in the analysis, or because of a changing business environment or other factors outside of the requirements engineer’s control.

Requirements engineering as a discipline must recognise the inevitability of change and devise the means to help organisations mitigate the effects of change. Whilst it is valid to stress the importance of analysis to get the requirements right in the first place, with the current state-of-the-art we do not know how to guarantee this. Just as structural engineering has evolved the means cope with poor ground conditions, so requirements engineering needs to cope with volatile requirements. This change has to be managed by applying careful requirements tracing, impact analysis and version management. Hence, the requirements engineering process is not merely a front-end task to software development, but spans the whole development life-cycle. In a typical project the activities of the requirements engineer evolve over time from elicitation to change management.

Requirements engineering is fundamentally an inter-disciplinary process. To adequately do their job, the requirements engineer needs to mediate between the domain of the system user (and other stakeholders) and the technical world of the software engineer. This requires that they have both technical skills and an understanding of the application domain. For example, the requirements engineer for an e-commerce system needs to understand e-commerce business models and digital signature legislation as well as the technical world of Internet solutions. Even for a skilled software engineer, application domain understanding has to be acquired through a combination of training, experience, and access to domain expertise. The job of eliciting requirements and reaching consensus on appropriate trade-offs also requires that the requirements engineer has strong inter-personal skills. They typically have to deal with people with different skills, roles and interests in the system, who may be unable to clearly articulate their requirements. One of the fundamental tenets of good software engineering is that there is good communication between system users and system developers. It is the requirements engineer who is the conduit for this communication.

3.6. Products and deliverables

Good requirements engineering requires that the products of the process - the deliverables - are defined. The most fundamental of these in requirements engineering is the requirements document. This often comprises two separate documents:

- A document that specifies the system requirements. This is sometimes known as the requirements definition document, user requirements document or, as defined by IEEE std 1362-1998, the concept of operations (ConOps) document. This document serves to define the high-level system requirements from the stakeholders' perspective(s). It also serves as a vehicle for validating the system requirements and, in certain types of project, may form the basis of an invitation to tender. Its typical readership includes representatives of the system stakeholders. It must be couched in terms of the customer's domain. In addition to a list of the system requirements, the requirements definition needs to include background information such as statements of the overall objectives for the system, a description of its target environment and a statement of the constraints and non-functional requirements on the system. It may include conceptual models designed to illustrate the system context, usage scenarios, the principal domain entities, and data, information and work flows.
- A document that specifies the software requirements. This is sometimes known as the software requirements specification (SRS). The purpose and readership of the SRS is somewhat different than the requirements definition document. In crude terms, the SRS documents the detailed requirements derived from elaboration of the system requirements, and which have been allocated to software. The non-functional requirements in the requirements definition should have been elaborated and quantified. The principal readership of the SRS is technical and this can be reflected in the language and notations used to describe the requirements, and in the detail of models used to illustrate the system. For custom software, the SRS may form the basis of a contract between the developer and customer.

This is only a broad characterisation of the requirements document(s) that may be mandated by a particular requirements engineering process. The essential point is that some medium is needed for communicating the requirements engineer’s assessment of the system requirements to the stakeholders, and the software requirements to developers. The requirements document must be structured to make information easy to find and standards such as IEEE std 1362-1998 and IEEE std 830-1998 provide guidance on this. Such standards are intended to be generic and need to be tailored to the context in which they are used.

A requirements document should be easy to read because this affects the likelihood that the system will conform to the requirements. It should also be reasonably modular so that it is easy to maintain. The structure of the requirements document contributes to these properties but care must also be taken to describe the requirements as precisely as possible.

Requirements are usually written in natural language but in the SRS this may be supplemented by formal or semi-formal descriptions. Selection of appropriate notations permits particular requirements and aspects of the system architecture to be described more precisely and concisely than natural language. The general rule is that notations should be used that allow the requirements to be described as precisely as possible. This is particularly crucial for safety-critical and certain other types of dependable systems. However, the choice of notation is often constrained by the training, skills and preferences of the document’s authors and readers.

In practice, it is seldom feasible to avoid natural language description. Natural language has many serious shortcomings as a medium for description. Among the most serious are that it is ambiguous and hard to describe complex concepts precisely. However, it is extraordinarily rich and able to describe, however imperfectly, almost any concept or system property. A natural language is also likely to be the document author and readerships’ only *lingua franca*. Because of this, requirements engineers must be trained to use language simply, concisely and to avoid common causes of mistaken interpretation. These include:

- long sentences with complex sub-clauses;
- the use of terms with more than one plausible interpretation (ambiguity);
- presenting several requirements as a single requirement;
- inconsistency in the use of terms.

To counteract these problems, requirements descriptions often adopt a stylised form and use a restricted subset of a natural language. It is good practice, for example, to keep requirement descriptions short and to standardise on a small set of modal verbs to indicate relative priorities. Hence, for example, the use of 'shall' in the requirement 'The emergency breaks shall be applied to bring the train to a stop if the nose of the train passes a signal at DANGER' indicates a requirement that is mandatory.

Verification of the quality of the requirements documents(s) is an essential part of requirements validation. Hence, requirements validation is not merely about checking that the requirements engineer has understood the requirements. It is also about checking that the way the requirements have been documented conforms to company standards, and is understandable, consistent and complete. The document(s) should be subjected to review by different stakeholders including representatives of the customer and developer. Crucially, requirements documents must be placed under the same configuration management regime as the other deliverables of the development process.

The requirements document(s) are only the most visible manifestation of the requirements. They exclude information that is not required by the document readership. However this other information is needed in order to manage them. In particular, it is essential that requirements are traced. Tracing refers to the construction of a directed asynchronous graph (DAG) that records the derivation of requirements and provides audit trails of requirements. As a minimum, requirements need to be tracable backwards to their source (e.g. from a software requirement back to the system requirements from which it was elaborated), and forwards to the design or implementation artifacts that implement them (e.g. from a software requirement to the design document for a component that implements it). Tracing allows the requirements to be managed. In particular, it allows an impact analysis to be performed for a proposed change to one of the requirements.

Requirements tracing and the maintenance of requirements attributes has historically been grossly under-valued. Part of the reason for this is that it is an overhead. However, modern requirements management tools make this much less so. They typically comprise a database of requirements and a graphical user interface:

- to store the requirement descriptions and attributes;
- to allow the trace DAGs to be generated automatically;
- to allow the propagation of requirements changes to be depicted graphically;
- to generate reports on the status of requirements (such as whether they have been analysed, approved, implemented, etc.);
- to generate requirements documents that conform to selected standards;
- and to apply version management to the requirements.

Contemporary recognition of the importance of requirements management owes much to its identification as a key practice area in level two of the capability maturity model for software (Software-CMM) [Pau95].

However, it should be noted that not every organisation has a culture of documenting and managing requirements. It is common for dynamic start-up companies which are driven by a strong 'product vision' and limited resources to view requirements documentation as an unnecessary overhead. Inevitably, however, as these companies expand, as their customer base grows and as their product starts to evolve, they discover that they need to recover the requirements that motivated product features in order to assess the impact of proposed changes. It is true that requirements documentation and management is an overhead, but it is one that pays dividends in the longer term.

4. Software requirements engineering breakdown: processes and activities

The knowledge area breakdown we have chosen is broadly compatible with the sections of ISO/IEC 12207-1995 that refer to requirements engineering activities. This standard views the software process at 3 different levels as primary, supporting and organisational life-cycle processes. In order to keep the breakdown simple, we conflate this structure into a single life-cycle process for requirements engineering. The separate topics that we *identify include primary life-cycle process activities such as requirements elicitation and requirements* analysis, along with requirements engineering-specific descriptions of management and, to a lesser degree, organisational processes. Hence, we identify *requirements validation* and *requirements management* as separate topics.

We are aware that a risk of this breakdown is that a waterfall-like process may be inferred. To guard against this, the first topic, *the requirements engineering process*, is designed to provide a high-level overview of requirements engineering by setting out the resources and constraints that requirements engineering operates under and which act to configure the requirements engineering process.

There are, of course, many other ways to structure the breakdown. For example, instead of a process-based structure, we could have used a product-based structure (system requirements, software requirements, prototypes, use-cases, etc.). We have chosen the process-based breakdown to reflect the fact that requirements engineering, if it is to be successful, must be considered as a process with complex, tightly coupled activities (both sequential and concurrent) rather than as a discrete, one-off activity at the outset of a software development project. We have substantial experience of this structure [Som97, Kot98] which is also compatible with other works on requirements engineering (see [Tha97], for example). See appendix B for an itemised rationale for the breakdown.

The breakdown comprises 6 topics:

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

Figure 1 is derived from [Boe94] and [Pot94] and shows conceptually, how these activities comprise an iterative requirements engineering process. The different activities in requirements engineering are repeated until an acceptable requirements specification document is produced or until external factors such as schedule pressure or lack of resources cause the requirements engineering process to terminate. After a final requirements document has been produced, any further changes become part of the requirements management process.

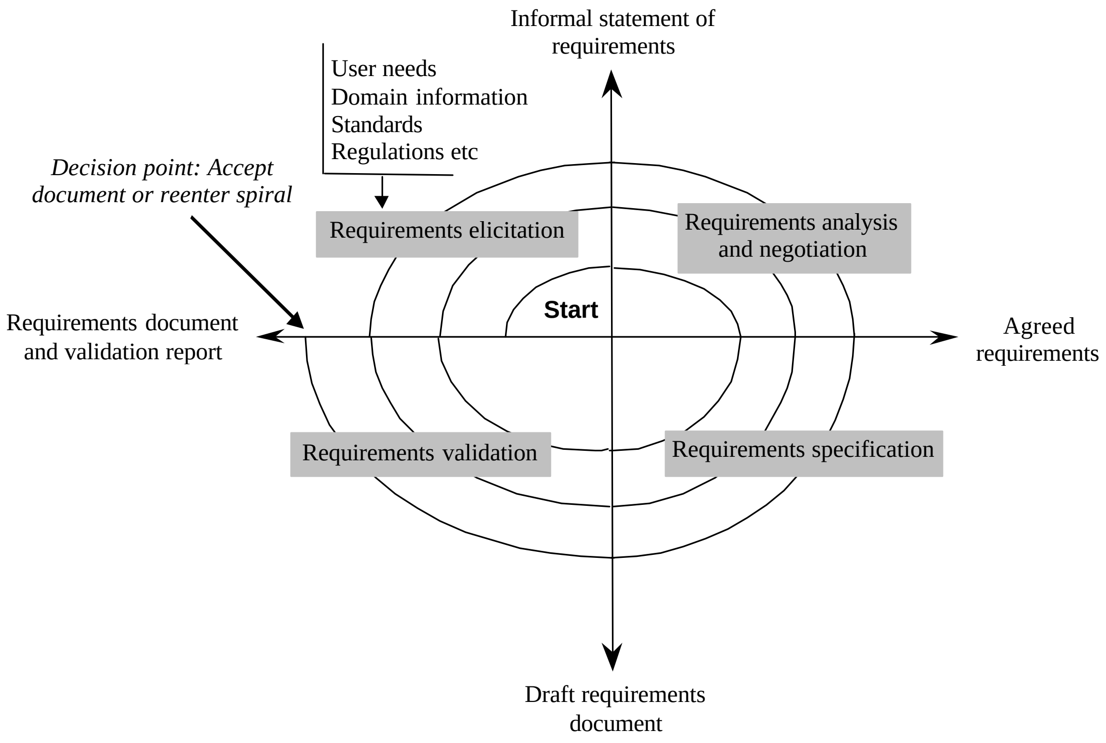
Informal statement of requirements
User needs
Domain information
Standards
Regulations etc
Decision point: Accept document or reenter spiral
Requirements elicitation
Requirements analysis and negotiation
Start
Requirements document and validation report
Agreed requirements
Requirements validation
Requirements specification
Draft requirements document

Figure 1 A spiral model of the requirements engineering process

4.1 The requirements engineering process

This section is concerned with introducing the requirements engineering process, orienting the remaining 5 topics and showing how requirements engineering dovetails with the overall software engineering process.

This section also deals with contractual and project organisation issues. The project organisation issues in this section are described with reference to the early phase in the project concerned with bounding system requirements to ensure that an achievable project is defined. The topic is broken down into 5 subtopics.

4.1.1 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;
- the need to manage requirements under the same configuration management regime as other products of the development process;
- will need to be tailored to the organisation and project context.

In particular, the subtopic shows how the activities of elicitation, analysis, specification, validation and management are configured for different types of project and constraints. It includes an overview of activities provide input to the process such as marketing and feasibility studies.

4.1.2 Process actors.
This subtopic introduces the roles of the people who participate in the requirements engineering process. Requirements engineering is fundamentally interdisciplinary and the requirements engineer needs to mediate between the domains of the user and software engineering. There are often many people involved besides the requirements engineer, each of whom have a stake in the system. The stakeholders will vary across different projects but always includes users/operators and customer (who need not be the same). These need not be homogeneous groups because there may be many users and many customers, each with different concerns. There may also be other stakeholders who are external to the user's/customer's organisation, such as regulatory authorities, who's requirements need to be carefully analysed. The system/software developers are also stakeholders because the have a legitimate interest in profiting from the system. Again, these may be a heterogeneous group in which (for example) the system architect has different concerns from the system tester.

It will not be possible to perfectly satisfy the requirements of every stakeholder and the requirements engineer's job is to negotiate a compromise that is both acceptable to the principal stakeholders and within budgetary, technical, regulatory and other constraints. A prerequisite for this is that all the stakeholders are indentified, the nature of their 'stake' is analysed and their requirements are elicited.

4.1.3 Process support and management.
This subtopic introduces the project management resources required and consumed by the requirements engineering process. This topic merely sets the context for topic 4 (Initiation and scope definition) of the software management KA. It's principal purpose is to make the link from process activities identified in 4.1.1 to issues of cost, human resources, training and tools.

4.1.4 Process quality and improvement.
This subtopic is concerned with requirements engineering process quality assessment. 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. This subtopic covers:

- requirements engineering coverage by process improvement standards and models;
- requirements engineering metrics and benchmarking;
- improvement planning and implementation;

Links to common themes	
Quality	The process quality and improvement subtopic is concerned with quality. It contains links to SPI standards such as the software and systems engineering CMMs [Pau95][SEI95], 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) [Som97].
Standards	SPI models/standards as above. In addition, the life-cycle software engineering standard ISO/IEC 12207-1995 describes software requirements engineering activities in the context of the primary, supporting and organisational life-cycle processes for software.
Measurement	Measurement is relatively difficult for many aspects of the requirementes engineering process (although metrics can be applied to the requirements document(s) – these are covered in the requirements specification subtopic). This is particularly true at the early stages where the artifacts that are being produced (requirements, models, etc.) are abstract and loosely connected to the quality of the end-product. Hence, requirements metrics tend to be relatively coarse-grained and concerned with (e.g.) counting numbers of requirements and numbers and effects of requirements changes.
Tools	General project management tools. Refer to the software management KA.

4.2 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.

4.2.1 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. A feasibility study is a relatively low-cost way of doing this.
- 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 (see 4.1.2). 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. 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.

4.2.2 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.
- 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 4.3.2. (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 is a wide 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 (4.5.2).
- Facilitated meetings. The purpose of these is to try to achieve a summative effect whereby a group of people can bring more insight to their requirements than by working individually. They can brainstorm and refine ideas that may be difficult to surface using (e.g.) interviews. Another advantage is that conflicting requirements are surfaced early on in a way that lets the stakeholders recognise where there is conflict. At its best, this technique may result in a richer and more consistent set of requirements than might otherwise be achievable. However, meetings need to be handled carefully (hence the need for a facilitator) to prevent phenomena such as ‘groupthink’ or the requirements reflecting the concerns of a few vociferous (and perhaps senior) people to the detriment of others.
- 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.

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. These typically set out the goals of elicitation but have little to say on techniques.
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.

4.3 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 system 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).

4.3.1 Requirements classification

There is a strong overlap between requirements classification and requirements attributes (4.6.2). Requirements can be classified on a number of dimensions (e.g. see [Har93]). Examples include:

- Whether the requirement is functional or non-functional (see 3.1).
- Whether the requirement is derived from one or more high-level requirements, an emergent property (see 3.4), or at a high level and imposed directly on the system by a stakeholder or some other source.
- Whether the requirement is on the product (functional or non-functional) or the process. Requirements on the process constrain, for example, the choice of contractor, the development practices to be adopted, and the standards to be adhered to.
- The requirement priority. In general, the higher the priority, the more essential the requirement is for meeting the overall goals of the system. Often classified on a fixed point scale such as *mandatory*, *highly desirable*, *desirable*, *optional*. In practice, priority often has to be balanced against cost of implementation.
- The scope of the requirement. Scope refers to the extent to which a requirement affects the system and system components. Some requirements, particularly certain non-functional ones, have a global scope in that their satisfaction cannot be allocated to a discrete component. Hence a requirement with global scope may strongly affect the system architecture and the design of many components, one with a narrow scope may offer a number of design choices with little impact on the satisfaction of other requirements.
- 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.

Other classifications may be appropriate, depending upon the development organization's normal practice and the application itself. Note that in all cases requirements must be unambiguously identified.

4.3.2 Conceptual modeling

The development of models of the problem is fundamental to requirements analysis (see 3.4). The purpose is to aid understanding of the problem rather than to initiate design of the solution. Hence, conceptual models comprise models of entities from the problem domain configured to reflect their real-world relationships and dependencies.

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 often more productive to adopt a modeling notation or method that the requirements engineer has experience with. However, it may be appropriate or necessary to adopt a notation that is better supported by tools, imposed as a process requirement (see 4.3.1), or simply ‘better’.
- 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.

4.3.3 Architectural design and requirements allocation

At some point the architecture of the solution must be derived. Architectural design is the point at which requirements engineering overlaps with software or systems design and illustrates how impossible it is to cleanly decouple both tasks. In many cases, the requirements engineer acts as system architect because the process of analysing and elaborating the requirements demands that the subsystems and components that will be responsible for satisfying the requirements be identified. This is requirements allocation – the assignment of responsibility for satisfying requirements to subsystems and components.

Allocation is important to permit detailed analysis of requirements. Hence, for example, once a set of requirements have been allocated to a component, they can be further analysed to discover requirements on how the component needs to interact with other components in order to satisfy the allocated requirements. In large projects, allocation stimulates a new round of analysis for each subsystem. As an example, requirements for a particular breaking performance for a car (breaking distance, safety in poor driving conditions, smoothness of application, pedal pressure required, etc.) may be allocated to the breaking hardware (meachanical and hydraulic assemblies) and an anti-lock breaking system (ABS). Only when a requirement for an anti-lock system has been identified, and the requirements are allocated to it can the capabilities of the ABS, the breaking hardware and emergent properties (such as the car weight) be used to identify the detailed ABS software requirements.

Architectural design is closely identified with conceptual modeling and in many cases it is a natural progression to derive the solution architecture from the domain architecture. There is not always a simple one-to-one mapping from real-world domain entities to computational components, however, so architectural design is identified as a separate sub-topic. The requirements of notations and methods are broadly the same for conceptual modeling and architectural design.

4.3.4 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. Detailed guidance is provided only by de-facto modeling 'standards' (e.g. SADT or UML) which may not be completely

	domain independent.
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.

4.4 Software requirements specification

This topic is concerned with the structure, quality and verification of the requirements document. This may take the form of two documents, or two parts of the same document with different readership and purposes (see 3.6): the requirements definition document and the software requirements specification. The topic stresses that documenting the requirements is the most fundamental precondition for successful requirements handling.

4.4.1 The requirements definition document

This document (sometimes known as the user requirements document or concept of operations) records the system requirements. It defines the high-level system requirements from the domain perspective. Its readership includes representatives of the system users/customers (marketing may play these roles for market-driven software) so it must be couched in terms of the domain. It must list the system requirements along with background information about the overall objectives for the system, its target environment and a statement of the constraints and non-functional requirements. It may include conceptual models designed to illustrate the system context, usage scenarios, the principal domain entities, and data, information and work flows.

4.4.2 The software requirements specification (SRS)

The SRS serves an important role in software systems development. Its benefits include:

- It establishes the basis for agreement between the customers and contractors or suppliers (in market-driven projects, these roles may be played by marketing and development divisions) on what the software product is to do and as well as what it should not do.
- It forces a rigorous assessment of requirements before design can begin and reduces later redesign.
- It provides a realistic basis for estimating product costs and schedules.
- Organisations can use a SRS to develop their own validation and verification plans more productively.
- Provides an informed a basis for transferring a software product to new users or new machines.
- Focuses on product rather than project and therefore provides a basis for product enhancement

4.4.3 Document structure and standards

This section describes the structure and content of a requirements document. It is also concerned with factors that influence how organisations interpret document standards to local circumstances. Several recommended guides and standards for SRS document exist. These include IEEE p123/D3 guide, IEEE Std. 1233 guide, IEEE std. 830-1998, ISO/IEC 12119-1994. IEEE std 1362-1998 *concept of operations* (ConOps) is a recent standard for a requirements definition document. Other guides and document template are also available in [Tha97].

4.4.4 Document quality

This section is concerned with assessing the quality of an SRS. This is one area where metrics can be usefully employed in requirements engineering. There are tangible attributes that can be measured. Moreover, the quality of the requirements document can dramatically affect the quality of the product.

A number of quality indicators have been developed that can be used to relate the quality of an SRS to other project variables such as cost, acceptance, performance, schedule, reproducibility etc. [Ros98]. Quality indicators for individual SRS statements include imperatives, directives, weak phrases, options and continuances. Indicators for the entire SRS document include size, readability, specification depth and text structure.

There is a strong overlap with 4.5.1 (the conduct of requirements reviews).

Links to common themes	
Quality	The quality of the requirements documents dramatically affects the quality of the product.
Standards	There are many of these. See 4.4.3.
Measurement	Quality attributes of requirements documents can be identified and measured. See 4.4.4.
Tools	Tool support for documentation exists in many forms from standard word processors to requirements management tools that may generate an SRS from their requirements database according to a standard template. Rudimentary quality checking tools are beginning to become commercially available. More sophisticated ones are being piloted in some organisations (see [Ros98] for an example).

4.5 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.

4.5.1 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 system requirements definition document, the software requirements specification document, the baseline specification for a new release, etc.

4.5.2 Prototyping.

Prototyping is commonly employed for validating the requirements engineer's interpretation of the system 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. For this reason, several people (e.g. [Rob99]) recommend prototypes that avoid software – such as flip-chart-based mockups. Prototypes may 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.

4.5.3 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).

4.5.4 Acceptance tests.

An essential property of a system 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.

Identifying and designing acceptance test may be difficult for non-functional requirements (see 3.1). To be verifiable, they must first be analysed to the point where they can be expressed quantitatively.

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 830-1998) 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.

4.6 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 3 subtopics concerned with requirements management.

4.6.1 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.

4.6.2 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 classification dimensions of 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.

4.6.3 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 system 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 system 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.

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 standards such as of ISO/IEC 12207-1995 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**
1. The requirement engineering process	Process models Process actors Process support and management Process quality and improvement
2. Requirements elicitation	Requirements sources Elicitation techniques
3. Requirement analysis	Requirements classification Conceptual modeling Architectural design and requirements allocation Requirements negotiation
4. Requirements specification	The requirements definition document The software requirements specification (SRS) Document structure and standards Document quality
5. Requirements validation	The conduct of requirements reviews Prototyping Model validation Acceptance tests
6. Requirements management	Change management Requirements attributes Requirements tracing

Appendix B Breakdown rationale

Criterion (a): Number of topic breakdowns
One breakdown provided

Criterion (b): Reasonableness
The breakdown is reasonable in that it covers the areas discussed in most requirements engineering texts and standards. However requirements validation is normally combined with requirements verification.

Criterion (c): Generally accepted
The breakdowns are generally accepted in that they cover areas typically in texts and standards.
At level A.1 the breakdown is identical to that given in most requirements engineering texts, apart from process improvement. Requirements engineering process improvement is an important emerging area in requirements engineering. We believe this topic adds great value to any the discussion of the requirements engineering as its directly concerned with process quality assessment.

At level A.2 the breakdown is identical to that given in most requirements engineering texts. At level A.3 the breakdown is similar to that discussed in most texts. We have incorporated a reasonably detailed section on requirement characterization to take into account the most commonly discussed ways of characterizing requirements [Har93]. A.4 the breakdown is similar to that discussed in most texts, apart from document quality assessment. We believe this an important aspect of the requirements specification document and deserves to be treated as a separate sub-section. In A.5 and A.6 the breakdown is similar to that discussed in most texts.

Criterion (d): No specific domains have been assumed
No specific domains have been assumed

Criterion (e): Compatible with various schools of though
Requirements engineering concept at the process level are general mature and stable.

Criterion (f): Compatible with industry, literature and standards
The breakdown used here has been derived from literature and relevant standards to reflect a consensus of opinion.

Criterion (g): As inclusive as possible
The inclusion of the requirements engineering process A.1 sets the context for all requirements engineering topics. This level is intended to capture the mature and stable concepts in requirements engineering. The subsequent levels all relate to level 1 but are general enough to allow more specific discussion or further breakdown.

Criterion (h): Themes of quality, tools, measurement and standards
The relationship of software requirements engineering product quality assurance, tools and standards is provided in the breakdown.

Criterion (i): 2 to 3 levels, 5 to 9 topics at the first level
The proposed breakdown satisfies this criterion.

Criterion (j): Topic names meaningful outside the guide
The topic names satisfy this criterion

Criterion (k): Vincenti breakdown
TBD

Criterion (l): Version 0.1 of the description

Criterion (m): Text on the rationale underlying the proposed breakdowns
This document provides the rationale

Appendix C 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
Process models			X	X				X	X	X	X
Process actors	X		X	X					X	X	X
Process support									X	X	X
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
Architectural design and requirements allocation	X				X		X	X	X	X	X
Requirements negotiation			X						X		
Requirement specification	X	X	X	X	X	X	X	X	X	X	X
The requirements definition document	X	X	X	X	X		X	X	X	X	X
The software requirements specification (SRS)	X	X	X	X	X		X	X	X	X	X
Document structure	X	X	X	X	X				X	X	X
Document quality	X		X								
Requirements validation	X	X		X				X	X	X	X
The conduct of 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

Table 1 Topics and their references

Appendix D - Software requirements engineering and Bloom's taxonomy

TOPIC	**Bloom Level**
Requirements engineering process	
Process models	Knowledge
Process actors	Knowledge
Process support	Knowledge
Process quality and improvement	Knowledge
Requirements elicitation	
Requirements sources	Comprehension
Elicitation techniques	Application
Requirements analysis	
Requirements classification	Comprehension
Conceptual modeling	Comprehension
Architectural design and requirements allocation	Analysis
Requirements negotiation	Analysis
Requirement specification	
The requirements definition document	Application
The software requirements specification (SRS)	Application
Document structure	Application
Document quality	Analysis
Requirements validation	
The conduct of requirements reviews	Analysis
Prototyping	Application
Model validation	Analysis
Acceptance tests	Application
Requirements management	
Change management	Analysis
Requirement attributes	Comprehension
Requirements tracing	Comprehension

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.

[Boe94] B. Boehm, P. Bose, et al., Software Requirements as Negotiated Win Conditions, Proc. 1st International Conference on Requirements Engineering (ICRE), Colorado Springs, Co, USA, pp 74-83, 1994.

[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

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

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

[Hut95] A. Hutchings and S. Knox, Creating products customers demand, Communications of the ACM, 38 (5), pp 72-80, May 1995.

[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.

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

[Pot94] C. Potts, K. Takahashi and A. Anton, Inquiry-Based Scenario Analysis of System Requirements, IEEE Software, 11, (2), 21-32.

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

[Rob99] S. Robertson, J. Robertson, *Mastering the requirements process*, Addison-Wesley, 1999.

[Ros98] L. Rosenberg, T.F. Hammer, and L.L. Huffman, Requirements, testing and metrics, 15th Annual Pacific Northwest Software Quality Conference, Utah, October 1998.

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

[SEI95] A Systems Engineering Capability Model, Version 1.1, CMU/SEI95-MM-003, Software Engineering Institute, 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

[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.

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