

REQUIREMENTS ANALYSIS*

Trung Hung VO

This work is produced by OpenStax-CN X and licensed under the
Creative Commons Attribution License 2.0[†]

Abstract

In this module, we'll discover effective techniques for managing the requirements engineering process all the way through the development cycle. It is concerned with the definition of software requirements and the products generated in that definition. The process involves requirements identification, requirements analysis, requirements specification, requirements communication and development of acceptance criteria and procedures.

1 Introduction

In systems engineering and software engineering, requirements analysis encompasses those tasks that go into determining the requirements of a new or altered system, taking account of the possibly conflicting requirements of the various stakeholders, such as users. Requirements analysis is critical to the success of a project.

Systematic requirements analysis is also known as requirements engineering. It is sometimes referred to loosely by names such as requirements gathering, requirements capture, or requirements specification. The term "requirements analysis" can also be applied specifically to the analysis proper (as opposed to elicitation or documentation of the requirements, for instance).

Requirements must be measurable, testable, related to identified business needs or opportunities, and defined to a level of detail sufficient for system design.

*Version 1.6: Jun 26, 2007 10:42 am -0500

[†]<http://creativecommons.org/licenses/by/2.0/>

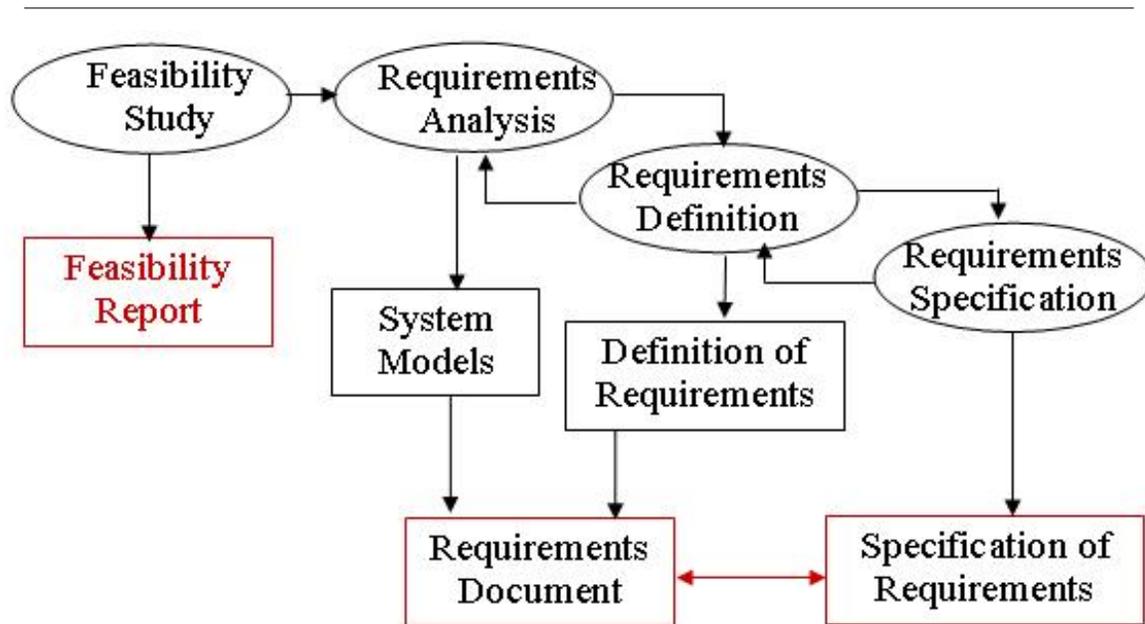


Figure 1: Requirements process

2 Software Requirements Fundamentals

2.1 Definition of a Software Requirement

At its most basic, a software requirement is a property which must be exhibited in order to solve some problem in the real world. This session refers to requirements on “software” because it is concerned with problems to be addressed by software. Hence, a software requirement is a property which must be exhibited by software developed or adapted to solve a particular problem. The problem may be to automate part of a task of someone who will use the software, to support the business processes of the organization that has commissioned the software, to correct shortcomings of existing software, to control a device, and many more. The functioning of users, business processes, and devices is typically complex. By extension, therefore, the requirements on particular software are typically a complex combination of requirements from different people at different levels of an organization and from the environment in which the software will operate.

An essential property of all software requirements is that they be verifiable. It may be difficult or costly to verify certain software requirements. For example, verification of the throughput requirement on the call center may necessitate the development of simulation software. Both the software requirements and software quality personnel must ensure that the requirements can be verified within the available resource constraints.

Requirements have other attributes in addition to the behavioral properties that they express. Common examples include a priority rating to enable trade-offs in the face of finite resources and a status value to enable project progress to be monitored. Typically, software requirements are uniquely identified so that they can be over the entire software life cycle.

2.2 Product and Process Requirements

A distinction can be drawn between product parameters and process parameters. Product parameters are requirements on software to be developed (for example, “The software shall verify that a student meets all prerequisites before he or she registers for a course.”).

A process parameter is essentially a constraint on the development of the software (for example, “The software shall be written in Ada.”). These are sometimes known as process requirements.

Some software requirements generate implicit process requirements. The choice of verification technique is one example. Another might be the use of particularly rigorous analysis techniques (such as formal specification methods) to reduce faults which can lead to inadequate reliability. Process requirements may also be imposed directly by the development organization, their customer, or a third party such as a safety regulator.

2.3 Functional and Non-functional Requirements

Functional requirements describe the functions that the software is to execute; for example, formatting some text or modulating a signal. They are sometimes known as capabilities or statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.

Nonfunctional requirements are the ones that act to constrain the solution. Nonfunctional requirements are sometimes known as constraints or quality requirements.

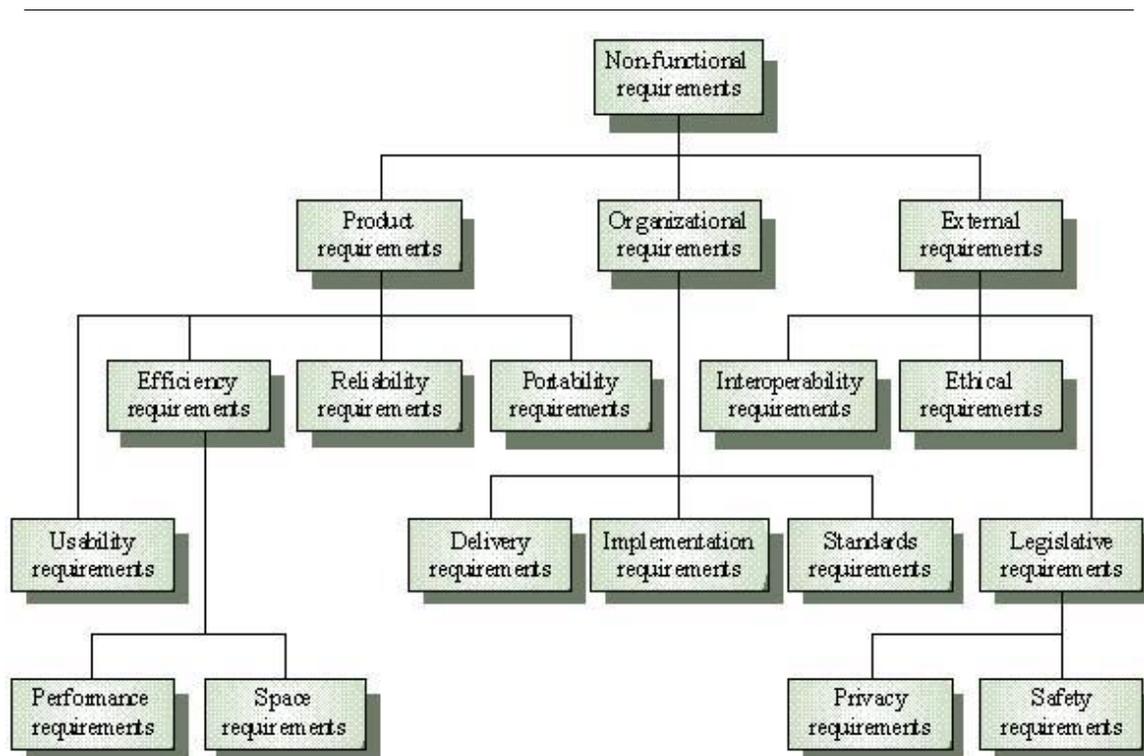


Figure 2: Nonfunctional Requirements

They can be further classified according to whether they are performance requirements, maintainability requirements, safety requirements, reliability requirements, or one of many other types of software requirements.

2.4 Emergent Properties

Some requirements represent emergent properties of software—that is, requirements which cannot be addressed by a single component, but which depend for their satisfaction on how all the software components interoperate. The throughput requirement for a call center would, for example, depend on how the telephone system, information system, and the operators all interacted under actual operating conditions. Emergent properties are crucially dependent on the system architecture.

2.5 Quantifiable Requirements

Software requirements should be stated as clearly and as unambiguously as possible, and, where appropriate, quantitatively. It is important to avoid vague and unverifiable requirements which depend for their interpretation on subjective judgment (“the software shall be reliable”; “the software shall be user-friendly”). This is particularly important for nonfunctional requirements. Two examples of quantified requirements are the following: a call center’s software must increase the center’s throughput by 20%; and a 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 used to derive a number of detailed requirements. The reliability requirement will tightly constrain the system architecture.

2.6 System Requirements and Software Requirements

In this topic, system means “an interacting combination of elements to accomplish a defined objective. These include hardware, software, firmware, people, information, techniques, facilities, services, and other support elements,” as defined by the International Council on Systems Engineering.

System requirements are the requirements for the system as a whole. In a system containing software components, software requirements are derived from system requirements.

The literature on requirements sometimes calls system requirements “user requirements.” We can define “user requirements” in a restricted way as the requirements of the system’s customers or end-users. System requirements, by contrast, encompass user requirements, requirements of other stakeholders (such as regulatory authorities), and requirements without an identifiable human source.

3 Requirements Process

This section introduces the software requirements process, orienting the remaining five subareas and showing how the requirements process dovetails with the overall software engineering process.

3.1 Process Models

The objective of this topic is to provide an understanding that the requirements process

- Is not a discrete front-end activity of the software life cycle, but rather a process initiated at the beginning of a project and continuing to be refined throughout the life cycle
- Identifies software requirements as configuration items, and manages them using the same software configuration management practices as other products of the software life cycle processes
- Needs to be adapted to the organization and project context

In particular, the topic is concerned with how the activities of elicitation, analysis, specification, and validation are configured for different types of projects and constraints.

3.2 Process Actors

This topic introduces the roles of the people who participate in the requirements process. This process is fundamentally interdisciplinary, and the requirements specialist needs to mediate between the domain of the stakeholder and that of software engineering. There are often many people involved besides the requirements specialist, each of whom has a stake in the software. The stakeholders will vary across projects, but always include users/operators and customers (who need not be the same).

Typical examples of software stakeholders include (but are not restricted to)

- Users: This group comprises those who will operate the software. It is often a heterogeneous group comprising people with different roles and requirements.
- Customers: This group comprises those who have commissioned the software or who represent the software's target market.
- Market analysts: A mass-market product will not have a commissioning customer, so marketing people are often needed to establish what the market needs and to act as proxy customers.
- Regulators: Many application domains such as banking and public transport are regulated. Software in these domains must comply with the requirements of the regulatory authorities.
- Software engineers: These individuals have a legitimate interest in profiting from developing the software by, for example, reusing components in other products. If, in this scenario, a customer of a particular product has specific requirements which compromise the potential for component reuse, the software engineers must carefully weigh their own stake against those of the customer.

It will not be possible to perfectly satisfy the requirements of every stakeholder, and it is the software engineer's job to negotiate trade-offs which are both acceptable to the principal stakeholders and within budgetary, technical, regulatory, and other constraints. A prerequisite for this is that all the stakeholders be identified, the nature of their "stake" analyzed, and their requirements elicited.

3.3 Process Support and Management

This topic introduces the project management resources required and consumed by the requirements process. It establishes the context for the first subarea (Initiation and scope definition) of the Software Engineering Management KA. Its principal purpose is to make the link between the process activities identified and the issues of cost, human resources, training, and tools.

3.4 Process Quality and Improvement

This topic is concerned with the assessment of the quality and improvement of the requirements process. Its purpose is to emphasize the key role the requirements process plays in terms of the cost and timeliness of a software product, and of the customer's satisfaction with it. It will help to orient the requirements process with quality standards and process improvement models for software and systems. Process quality and improvement is closely related to both the Software Quality KA and the Software Engineering Process KA. Of particular interest are issues of software quality attributes and measurement, and software process definition. This topic covers

- Requirements process coverage by process improvement standards and models
- Requirements process measures and benchmarking
- Improvement planning and implementation

4 Requirements Elicitation

Requirements elicitation is concerned with where software requirements come from and how the software engineer can collect them. It 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 and the customer. It is variously termed “requirements capture,” “requirements discovery,” and “requirements acquisition.”

One of the fundamental tenets of good software engineering is that there be good communication between software users and software engineers. Before development begins, requirements specialists may form the conduit for this communication. They must mediate between the domain of the software users (and other stakeholders) and the technical world of the software engineer.

4.1 Requirements Sources

Requirements have many sources in typical software, and it is essential that all potential sources be identified and evaluated for their impact on it. This topic is designed to promote awareness of the various sources of software requirements and of the 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 software. Goals provide the motivation for the software, but are often vaguely formulated. Software engineers need to pay particular attention to assessing the value (relative to priority) and cost of goals. A feasibility study is a relatively low-cost way of doing this.
- **Domain knowledge:** The software engineer needs to acquire, or have available, knowledge about the application domain. This enables them to infer tacit knowledge that the stakeholders do not articulate, assess the trade-offs that will be necessary between conflicting requirements, and, sometimes, to act as a “user” champion.
- **Stakeholders:** Much software has proved unsatisfactory because it has stressed the requirements of one group of stakeholders at the expense of those of others. Hence, software is delivered which is difficult to use or which subverts the cultural or political structures of the customer organization. The software engineer needs to identify, represent, and manage the “viewpoints” of many different types of stakeholders.
- **The operational environment:** Requirements will be derived from the environment in which the software will be executed. These may be, for example, timing constraints in real-time software or interoperability constraints in an office environment. These must be actively sought out, because they can greatly affect software feasibility and cost, and restrict design choices.
- **The organizational environment:** Software is often required to support a business process, the selection of which may be conditioned by the structure, culture, and internal politics of the organization. The software engineer needs to be sensitive to these, since, in general, new software should not force unplanned change on the business process.

4.2 Elicitation Techniques

Once the requirements sources have been identified, the software engineer can start eliciting requirements from them. This topic concentrates on techniques for getting human stakeholders to articulate their requirements. It is a very difficult area and the software 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 software engineer has to work hard to elicit the right information. A number of techniques exist for doing this, the principal ones being,

- **Interviews:** a “traditional” means of eliciting requirements. It is important to understand the advantages and limitations of interviews and how they should be conducted.
- **Scenarios:** a valuable means for providing context to the elicitation of user requirements. They allow the software engineer to provide a framework for questions about user tasks by permitting “what if” and “how is this done” questions to be asked. The most common type of scenario is the use case.

- Prototypes: a valuable tool for clarifying unclear requirements. They can act in a similar way to scenarios by providing users with a context within which they can better understand what information they need to provide. There is a wide range of prototyping techniques, from paper mock-ups of screen designs to beta-test versions of software products, and a strong overlap of their use for requirements elicitation and the use of prototypes for requirements validation.
- Facilitated meetings: The purpose of these is to try to achieve a summative effect whereby a group of people can bring more insight into their software requirements than by working individually. They can brainstorm and refine ideas which may be difficult to bring to the surface using interviews. Another advantage is that conflicting requirements surface early on in a way that lets the stakeholders recognize where there is conflict. When it works well, 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 a situation from occurring where the critical abilities of the team are eroded by group loyalty, or the requirements reflecting the concerns of a few outspoken (and perhaps senior) people are favored to the detriment of others.
- Observation: The importance of software context within the organizational environment has led to the adaptation of observational techniques for requirements elicitation. Software engineers learn about user tasks by immersing themselves in the environment and observing how users interact with their software and with each other. These techniques are relatively expensive, but they are instructive because they illustrate that many user tasks and business processes are too subtle and complex for their actors to describe easily.

5 Requirements Analysis

This topic is concerned with the process of analyzing requirements to

- Detect and resolve conflicts between requirements
- Discover the bounds of the software and how it must interact with its environment
- Elaborate system requirements to derive software requirements

The traditional view of requirements analysis has been that it be reduced to conceptual modeling using one of a number of analysis methods such as the Structured Analysis and Design Technique (SADT). 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).

Care must be taken to describe requirements precisely enough to enable the requirements to be validated, their implementation to be verified, and their costs to be estimated.

5.1 Requirements Classification

Requirements can be classified on a number of dimensions:

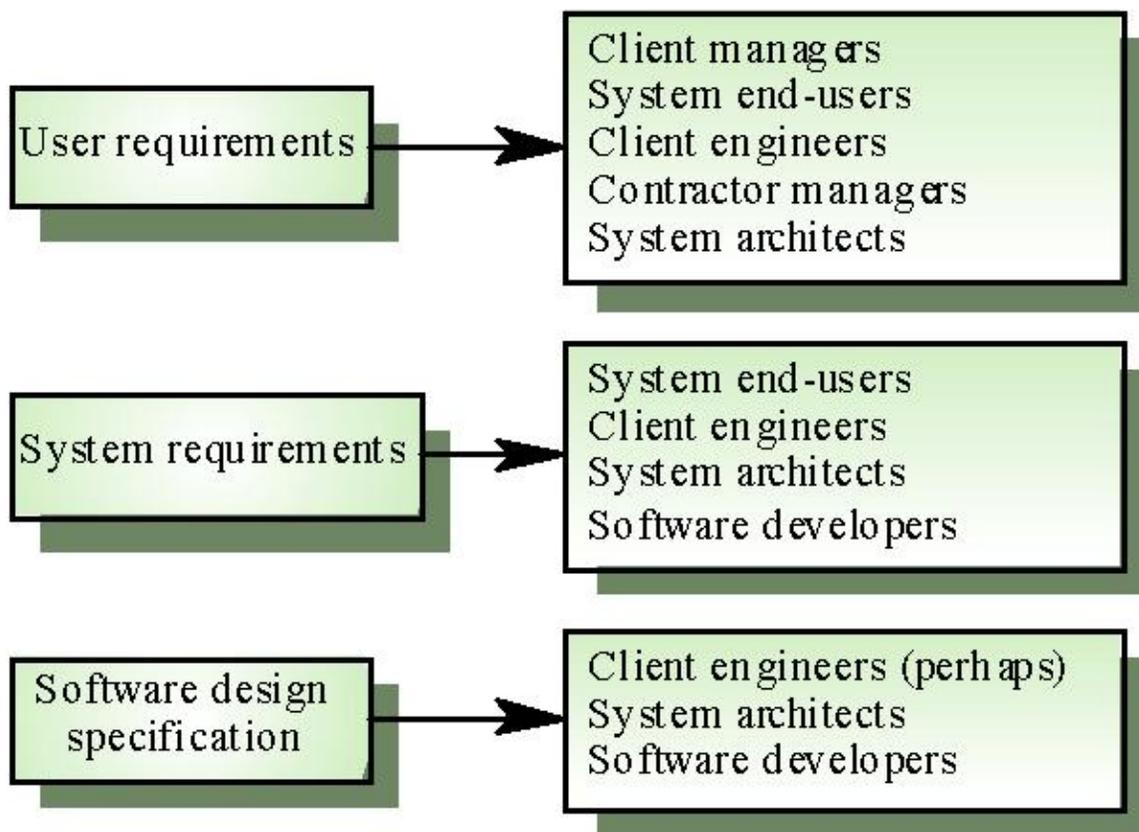


Figure 3: Requirement types

Other classifications may be appropriate, depending upon the organization's normal practice and the application itself.

There is a strong overlap between requirements classification and requirements attributes.

5.2 Conceptual Modeling

The development of models of a real-world problem is key to software requirements analysis. Their purpose is to aid in understanding 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.

Several kinds of models can be developed. These include data and control flows, state models, event traces, user interactions, object models, data models, and many others. The factors that influence the choice of model include

- The nature of the problem. Some types of software demand that certain aspects be analyzed particularly rigorously. For example, control flow and state models are likely to be more important for real-time software than for management information software, while it would usually be the opposite for data models.

- The expertise of the software engineer. It is often more productive to adopt a modeling notation or method with which the software engineer has experience.
- The process requirements of the customer. Customers may impose their favored notation or method, or prohibit any with which they are unfamiliar. This factor can conflict with the previous factor.
- The availability of methods and tools. Notations or methods which are poorly supported by training and tools may not achieve widespread acceptance even if they are suited to particular types of problems.

Note that, in almost all cases, it is useful to start by building a model of the software context. The software context provides a connection between the intended software and its external environment. This is crucial to understanding the software's context in its operational environment and to identifying its interfaces with 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 which guides the application of the notations. There is little empirical evidence to support claims for the superiority of one notation over another. However, the widespread acceptance of a particular method or notation can lead to beneficial industry-wide pooling of skills and knowledge. This is currently the situation with the UML (Unified Modeling Language).

Formal modeling using notations based on discrete mathematics, and which are traceable 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.

Two standards provide notations which may be useful in performing conceptual modeling—IEEE Std 1320.1, IDEF0 for functional modeling; and IEEE Std 1320.2, IDEF1X97 (IDEFObject) for information modeling.

5.3 Architectural Design and Requirements Allocation

At some point, the architecture of the solution must be derived. Architectural design is the point at which the requirements process overlaps with software or systems design and illustrates how impossible it is to cleanly decouple the two tasks. This topic is closely related to the Software Structure and Architecture subarea in the Software Design KA. In many cases, the software engineer acts as software architect because the process of analyzing and elaborating the requirements demands that the components that will be responsible for satisfying the requirements be identified. This is requirements allocation—the assignment, to components, of the responsibility for satisfying requirements.

Allocation is important to permit detailed analysis of requirements. Hence, for example, once a set of requirements has been allocated to a component, the individual requirements can be further analyzed to discover further 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.

Architectural design is closely identified with conceptual modeling. The mapping from real-world domain entities to software components is not always obvious, so architectural design is identified as a separate topic. The requirements of notations and methods are broadly the same for both conceptual modeling and architectural design.

5.4 Requirements Negotiation

Another term commonly used for this sub-topic is “conflict resolution.” This concerns resolving problems with requirements where conflicts occur between two stakeholders requiring mutually incompatible features, between requirements and resources, or between functional and non-functional requirements. In most cases, it is unwise for the software engineer to make a unilateral decision, and so it becomes 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 be traceable back to the customer. We have classified this as a software requirements analysis topic because problems emerge as the result of analysis. However, a strong case can also be made for considering it a requirements validation topic.

5.5 Requirements Specification

For most engineering professions, the term “specification” refers to the assignment of numerical values or limits to a product’s design goals. Typical physical systems have a relatively small number of such values. Typical software has a large number of requirements, and the emphasis is shared between performing the numerical quantification and managing the complexity of interaction among the large number of requirements. So, in software engineering jargon, “software requirements specification” typically refers to the production of a document, or its electronic equivalent, which can be systematically reviewed, evaluated, and approved.

For complex systems, particularly those involving substantial non-software components, as many as three different types of documents are produced: system definition, system requirements, and software requirements. For simple software products, only the third of these is required.

There are some approaches to requirements specification:

- Natural language
- Structured natural language
- Design description language
- Requirements specification language
- Graphical notation
- Formal specification

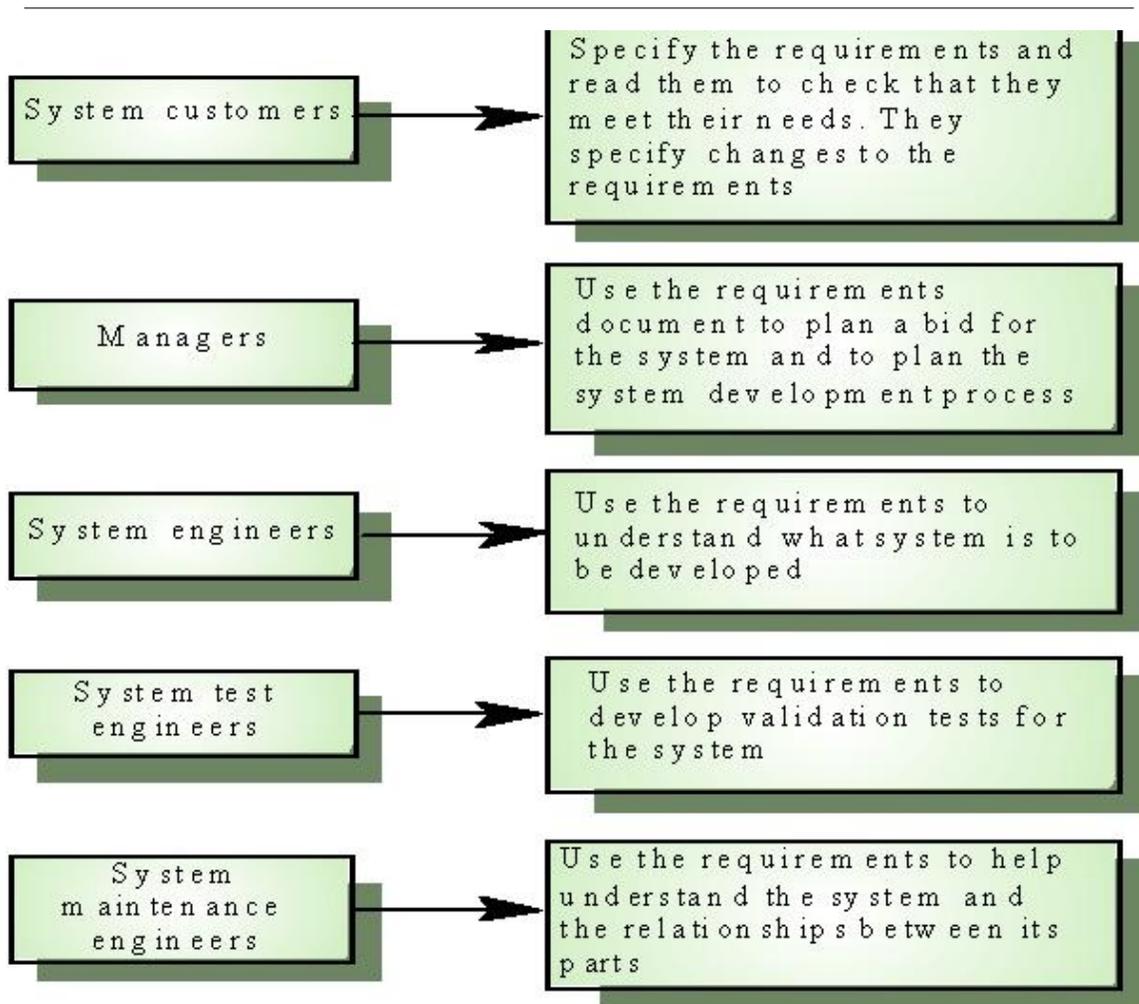


Figure 4: Types of requirement document

5.6 The System 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 its content must be couched in terms of the domain. The document lists the system requirements along with background information about the overall objectives for the system, its target environment and a statement of the constraints, assumptions, and non-functional requirements. It may include conceptual models designed to illustrate the system context, usage scenarios and the principal domain entities, as well as data, information, and workflows. IEEE Std 1362, Concept of Operations Document, provides advice on the preparation and content of such a document. (IEEE1362-98)

5.7 System Requirements Specification

Developers of systems with substantial software and non-software components, a modern airliner, for example, often separate the description of system requirements from the description of software requirements. In this view, system requirements are specified, the software requirements are derived from the system requirements, and then the requirements for the software components are specified. Strictly speaking, system requirements specification is a systems engineering activity and falls outside the scope of this Guide. IEEE Std 1233 is a guide for developing system requirements.

5.8 Software Requirements Specification

Software requirements specification establishes the basis for agreement between customers and contractors or suppliers (in market-driven projects, these roles may be played by the marketing and development divisions) on what the software product is to do, as well as what it is not expected to do. For non-technical readers, the software requirements specification document is often accompanied by a software requirements definition document.

Software requirements specification permits a rigorous assessment of requirements before design can begin and reduces later redesign. It should also provide a realistic basis for estimating product costs, risks, and schedules.

Organizations can also use a software requirements specification document to develop their own validation and verification plans more productively.

Software requirements specification provides an informed basis for transferring a software product to new users or new machines. Finally, it can provide a basis for software enhancement.

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

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

6 Requirements validation

The requirements documents may be subject to validation and verification procedures. The requirements may be validated to ensure that the software engineer has understood the requirements, and it is also important to verify that a requirements document conforms to company standards, and that it is understandable, consistent, and complete. Formal notations offer the important advantage of permitting the last two properties to be proven (in a restricted sense, at least). Different stakeholders, including representatives of the customer and developer, should review the document(s). Requirements documents are subject to the same software configuration management practices as the other deliverables of the software life cycle processes.

It is normal to explicitly schedule one or more points in the requirements process where the requirements are validated. The aim is to pick up any problems before resources are committed to addressing the requirements. Requirements validation is concerned with the process of examining the requirements document to ensure that it defines the right software (that is, the software that the users expect).

6.1 Requirements Reviews

Perhaps the most common means of validation is by inspection or reviews of the requirements document(s). A group of reviewers is assigned 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 definition document, the system specification document, the software requirements specification document, the baseline specification for a new release, or at any other step in the process.

6.2 Prototyping

Prototyping is commonly a means for validating the software engineer's interpretation of the software 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 where prototype validation may be appropriate. The advantage of prototypes is that they can make it easier to interpret the software engineer's assumptions and, where needed, give useful feedback on why they are wrong. For example, the dynamic behavior 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 recommend prototypes which avoid software, such as flip-chart-based mock-ups. Prototypes may be costly to develop. However, if they avoid the wastage of resources caused by trying to satisfy erroneous requirements, their cost can be more easily justified.

6.3 Model Validation

It is typically necessary to validate the quality of the models developed during analysis. For example, in object models, it is useful to perform a static analysis to verify that communication paths exist between objects which, in the stakeholders' domain, exchange data. If formal specification notations are used, it is possible to use formal reasoning to prove specification properties.

6.4 Acceptance Tests

An essential property of a software requirement is that it should be possible to validate that the finished product satisfies it. Requirements which cannot be validated are really just "wishes." An important task is therefore planning how to verify each requirement. In most cases, designing acceptance tests does this.

Identifying and designing acceptance tests may be difficult for non-functional requirements. To be validated, they must first be analyzed to the point where they can be expressed quantitatively.

7 Practical Considerations

The first level of decomposition of subareas presented in this KA may seem to describe a linear sequence of activities. This is a simplified view of the process.

The requirements process spans the whole software life cycle. Change management and the maintenance of the requirements in a state which accurately mirrors the software to be built, or that has been built, are key to the success of the software engineering process.

Not every organization has a culture of documenting and managing requirements. It is frequent in dynamic start-up companies, driven by a strong "product vision" and limited resources, to view requirements documentation as an unnecessary overhead. Most often, 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. Hence, requirements documentation and change management are key to the success of any requirements process.

7.1 Iterative Nature of the Requirements Process

There is general pressure in the software industry for ever shorter development cycles, and this is particularly pronounced in highly competitive market-driven sectors. Moreover, most projects are constrained in some way by their environment, and many are upgrades to, or revisions of, existing software where the architecture is a given. In practice, therefore, it is almost always impractical to implement the requirements process as a linear, deterministic process in which software requirements are elicited from the stakeholders, baselined, allocated, and handed over to the software development team. It is certainly a myth that the requirements for large software projects are ever perfectly understood or perfectly specified.

Instead, requirements typically iterate towards a level of quality and detail which is sufficient to permit design and procurement decisions to be made. In some projects, this may result in the requirements being baselined before all their properties are fully understood. This risks expensive rework if problems emerge late in the software engineering process. However, software engineers are necessarily constrained by project management plans and must therefore take steps to ensure that the “quality” of the requirements is as high as possible given the available resources. They should, for example, make explicit any assumptions which underpin the requirements, as well as any known problems.

In almost all cases, requirements understanding continues to evolve as design and development proceeds. This often leads to the revision of requirements late in the life cycle. Perhaps the most crucial point in understanding requirements engineering is that a significant proportion of the requirements will change. This is sometimes due to errors in the analysis, but it is frequently an inevitable consequence of change in the “environment”: for example, the customer’s operating or business environment, or the market into which software must sell. Whatever the cause, it is important to recognize the inevitability of change and take steps to mitigate its effects. Change has to be managed by ensuring that proposed changes go through a defined review and approval process, and, by applying careful requirements tracing, impact analysis, and software configuration management. Hence, the requirements process is not merely a front-end task in software development, but spans the whole software life cycle. In a typical project, the software requirements activities evolve over time from elicitation to change management.

7.2 Change Management

Change management is central to the management of requirements. This topic describes the role of change management, the procedures that need to be in place, and the analysis that should be applied to proposed changes. It has strong links to the Software Configuration Management KA.

7.3 Requirements Attributes

Requirements should consist not only of a specification of what is required, but also of ancillary information which helps manage and interpret the requirements. This should include the various classification dimensions of the requirement 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 important requirements attribute, however, is an identifier which allows the requirements to be uniquely and unambiguously identified.

7.4 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 which 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 the 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).

7.5 Measuring Requirements

As a practical matter, it is typically useful to have some concept of the “volume” of the requirements for a particular software product. This number is useful in evaluating the “size” of a change in requirements, in estimating the cost of a development or maintenance task, or simply for use as the denominator in other measurements.

Property	Measure
Speed	Processed transactions/second User/Event response time Screen refresh time
Size	K Bytes Number of RAM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Figure 5: Requirement measurements