Chapter 4
Writing and Reviewing Requirements

To write simply is as difficult as to be good.

William Somerset Maugham, author, 1874–1965 AD

4.1 Introduction

Requirements engineering is a technical process. Writing requirements is therefore not like other kinds of writing. It is certainly not like writing a novel, or a book like this; it is not even like the kind of "technical writing" seen in instruction manuals and user guides.

The purpose of this chapter is to present those aspects of writing requirements that are common to every development layer. Wherever the generic process is instantiated, certain principles and techniques are constant in their application to the expression and structuring of requirements.

In writing a requirements document, two aspects have to be carefully balanced:

1. The need to make the requirements document readable
2. The need to make the set of requirements processable

The first of these concerns the structure of the document, how it is organised and how the flow of it helps the reviewer to place individual requirement statements into context. The second focuses on the qualities of individual statements of requirement, the language used to promote clarity and preciseness, and how they are divided into single traceable items.

The experienced requirements engineer comes to realise that a word processor alone is not sufficient to manage a set of requirements, for the individual statements need to be identified, classified and traced. A classic problem, for instance, is the use of paragraph numbers to identify requirements: insert a new one in the middle, and suddenly all the subsequent requirement identifiers have changed.

Equally, those who have tried simply to manage their requirements in a database quickly realise that tables full of individual statements are unmanageable. Despite having the ability to identify, classify and sort requirements, vital contextual information

E. Hull et al., *Requirements Engineering*, DOI 10.1007/978-1-84996-405-0_4,

provided by the document has been lost; single statements lose meaning when separated from their place in the whole.

So both aspects – document and individuality – need to be maintained.

The writing and the reviewing of requirements (or any other kind of document, for that matter) should go hand-in-hand, in that the criteria for writing a good requirement are exactly those criteria against which the requirement should be reviewed. Hence the subjects are treated together in this chapter.

4.2 Requirements for Requirements

Before discussing how requirements documents and statements should be written, it is best to review some of the objectives and purpose for the writing of requirements in the first place. This will help in understanding why certain principles are suggested.

The starting place is the identification of stakeholders, which is shown in Table 4.1.

Table 4.1 Stakeholders for requirements

Stakeholder	Role
Author	Creates the requirements and incorporates changes
Publisher	Issues and archives the requirements document
Reviewer	Reviews the requirements and suggests changes
Implementer	Analyses the requirements and negotiates changes

Table 4.2 Abilities required for requirements

Ability
Ability uniquely to identify every statement of requirement.
Ability to classify every statement of requirement in multiple ways, such as: • By importance • By type (e.g. functional, performance, constraint, safety) • By urgency (when it has to be provided)
Ability to track the status of every statement of requirement, in support of multiple processes, such as: • Review status • Satisfaction status • Qualification status
Ability to elaborate a requirement in multiple ways, such as by providing: • Performance information • Quantification • Test criteria • Rationale • Comments
Ability to view a statement of requirement in the document context, i.e. alongside its surrounding statements.
Ability to navigate through a requirements document to find requirements according to a particular classification or context.
Ability to trace to any individual statement of requirement.

Table 4.2 lists capabilities required by the various stakeholders that relate to how requirements documents and statements are written. These are the basic things that one needs to be able to do to – and with – requirements, including identification, classification, elaboration, tracking status, tracing, placing in context and retrieving. How requirements are expressed and organised has a great influence on how "useable" the sets of requires becomes.

4.3 Structuring Requirements Documents

Requirements documentation can be very large. On paper, the complete subsystem requirements for an aircraft carrier, for instance, may fill many filing cabinets. It is not unknown for supplier responses to large systems to be delivered in lorries. In such situations, having a well-understood, clearly documented structure for the whole requirements set is essential to the effective management of complexity.

Organising requirements into the right structure can help:

- *Minimize* the number of requirements
- *Understand* large amounts of information
- *Find* sets of requirements relating to particular topics
- *Detect* omissions and duplications
- *Eliminate* conflicts between requirements
- *Manage* iteration (e.g. delayed requirements)
- *Reject* poor requirements.
- *Evaluate* requirements
- *Reuse* requirements across projects

Documents are typically hierarchical, with sections and subsections to multiple levels. Hierarchies are useful structures for classification, and one way of structuring a requirements document is to use the section heading structure to categorise the requirements statements. In such a regime, the position a requirement statement has in the document represents its primary classification. (Secondary classifications can be given through links to other sections, or by using attributes.)

Chapter 3 describes how system models frequently use hierarchies in the analysis of a system. Examples are:

Goal or capability decomposition as in stakeholder scenarios
Functional decomposition as in data-flow diagrams
State decomposition as in state charts

Where requirements are derived from such models, one of the resulting hierarchies can be used as part of the heading structure for the requirements document.

In addition to requirements statements themselves, requirements documents may contain a variety of technical and non-technical text, which support the understanding of the requirements. These may be such things as:

- *Background information* that places the requirements in context
- *External context* describing the enclosing system, often called "domain knowledge"
- *Definition of the scope* of the requirements (what's in and what's out)
- *Definitions of terms* used in the requirement statements
- *Descriptive text* which bridges different sections of the document
- *Stakeholder descriptions*
- *Summary of models* used in deriving the requirements
- *References* to other documents

4.4 Key Requirements

Many organisations use the concept of "key requirements", particularly at the stakeholder level. Often referred to as KURs ("Key User Requirements") or KPIs ("Key Performance Indicators"), these requirements are a small subset abstracted from the whole that capture the essence of the system.

The guiding philosophy when selecting key requirements is similar to that used by Jerome K. Jerome's "Three Men in a Boat", who, when planning for the trip, realised that

> *the upper reaches of the Thames would not allow the navigation of a boat sufficiently large to take the things [they] had set down as indispensable. …*
>
> *George said, 'We must not think of the things we could do with,but only the things that we cannot do without.'*

Every key requirement should solicit a negative response to the question:

> *If the solution didn't provide me with this capability, would I still buy it?*

or, if at the system level,

> *If the system didn't do this, would I still want it?*

In this way, the key requirements become those that are absolutely mandatory. (Of course, everything is negotiable, but trading key requirements would always engender very careful consideration.)

Where appropriate, each key requirement should be quantified with performance attributes. Doing this allows them to be used as KPIs, used to assess alternative proposals against the requirements, or used as a summary of vital statistics on project progress.

4.5 Using Attributes

It is clear from the discussions of process in previous chapters, and from the list of abilities in Table 4.2, that a simple textual statement is not sufficient fully to define a requirement; there is other classification and status information that each requirement carries.

Rather than clutter the text of a requirement, additional information should be placed in "attributes" attached to the requirement. Attributes allow the information associated with a single requirement to be structured for ease of processing, filtering, sorting, etc. Attributes can be used to support many of the abilities in Table 4.2, enabling the requirements to be sorted or selected for further action, and enabling the requirements development process itself to be controlled. Figure 4.1 shows an example of a requirement with a number of attributes.

The particular attributes used will depend on the exact processes that need to be supported. Some attributes are entirely automatic – e.g. dates, numbers – some come from users – e.g. priority – other attributes are flags, which are set after analysis work – e.g. checkability.

The following suggestions for attribute categories are drawn in part from some work carried out by a requirements working group in the UK chapter of INCOSE (Table 4.3).

[SH234] The ambulance control system shall be able to handle up to 100 simultaneous emergency calls.

Source:	R. Thomas
Priority:	Mandatory
Release:	1
Review status:	Accepted
Verifiable:	Yes
Verification:	By simulation, then by system test.

Fig. 4.1 Requirements attributes

Table 4.3 Categories of Attributes

Category	Example Values
Identification	
• Identifier	Unique reference
• Name	Unique name summarising the subject of the requirement.
Intrinsic characteristics	
• Basic type	Functional, performance, quality factor, environment, interface, constraint, non-requirement
• Quality factor sub-type	Availability, flexibility, integrity, maintainability, portability, reliability, safety, security, supportability, sustainability, usability, workmanship
• Product/process type	Product, process, data, service
• Quantitative/qualitative type	Quantitative, qualitative
• Life-cycle phase	Pre-concept, concept, development, manufacturing, integration/test, deployment/delivery/installation, operation, support, disposal
Priority and importance	
• Priority (compliance level)	Key, mandatory, optional, desirable *or* Must, Should, Could, Would (MoSCoW)
• Importance	1–10

(continued)

Table 4.3 (continued)

Category	Example Values
Source and ownership	
• Derivation type	Allocation, decomposition
• Source (origin)	Name of document or stakeholder
• Owner	Name of stakeholder
• Approval authority	Name or person
Context	
• Requirements set/document • Subject • Scope	(Best handled through positioning the requirement in a structured document.)
Verification and validation	
• V&V method	Analysis, inspection, system test, component test
• V&V stage	(See life-cycle phase.)
• V&V status	Pending, pass, failed, inconclusive
• Satisfaction argument	Rationale for choice of decomposition
• Validation argument	Rationale for choice of V&V methods
Process support	
• Agreement status	Proposed, being assessed, agreed
• Qualification status	Not qualified, qualified, suspect
• Satisfaction status	Not satisfied, satisfied, suspect
• Review status	To be reviewed, Accepted, Rejected
Elaboration	
• Rationale	Textual statement about why the requirement is present.
• Comments	Textual comments of clarification.
• Questions	Questions to be posed for clarification.
• Responses	Responses received for clarification.
Miscellaneous	
• Maturity (stability)	Number of changes/time
• Risk level	High, Medium, Low
• Estimated cost	
• Actual cost	
• Product release	Version(s) of product meeting the requirement.

4.6 Ensuring Consistency Across Requirements

A frequent concern in managing large sets of requirements is being able to identify conflicting requirements. The difficulty is in spotting that two statements many pages apart are in conflict. What techniques can be applied to assist in identifying these potential inconsistencies?

One answer lies in classifying requirements in several ways, and using filtering and sorting techniques to draw together small numbers of statements that address the same topic. Many requirements will touch on several aspects of a system. For instance, a requirement primarily about engine performance may also contain a safety element. Such a statement should therefore be viewed in both an engine performance context as well as in a safety context.

To facilitate this, requirements can be given primary and secondary classifications, as discussed in Section 1.3. Typically, each has a single primary classification (perhaps by virtue of its position in the document), and multiple secondary classifications, perhaps using links or attributes.

A thorough review process can now include the systematic filtering of statements by keywords used in primary and secondary classifications. For example, filtering on all requirements to do with safety will draw together statements whose primary classifications may be quite diverse. These can then be reviewed in proximity for potential conflicts.

4.7 Value of a Requirement

Some requirements are non-negotiable. If they are not met, the product is of no use.

Other requirements are negotiable. For instance, if a system is required to support at least 100 simultaneous users, but the delivered solution only supports 99, then it is most likely still of some value to the customer.

Capturing the value of a requirement can be a challenge. A way needs to be found of expressing the idea that, while the target may be 100 simultaneous users, 75 would be acceptable, but anything less than 50 is not acceptable; and maybe 200 would be even better.

One approach to this is to provide several performance values. Here is an example of a three-valued approach:

M: the mandatory lower (or upper) limit
D: the desired value
B: and the best value

These three values can be held in separate attributes, or represented within the text in a labelled form, such as "The system shall support [M:50, D:100, B:200] simultaneous users."

Another approach is to represent the value of a requirement by supplying a function that maps performance to some representation of value, usually a figure between 1 and 100. Figure 4.2 shows four examples of different shapes of value function. Function (a) shows the example above, where the number of simultaneous users should be maximised, but more than a minimum number is mandatory. Function (b) is the binary case: either the performance of 100 is exceeded or not. A performance of 200 does not add extra value. Function (c) shows a performance that is to be minimised (weight, for instance), whereas (d) shows one that is to be optimised (engine revs, for example).

This is a very visual way of presenting value. One glance at the shape of the value curve indicates the nature of the requirement: minimise, maximise, optimise, etc. It also allows the engineers to understand the degrees of freedom they have in designing solutions that deliver the best overall value, by trading-off performance between requirements. This is why this approach is frequently used as part of the tender assessment process, to judge between the relative values of alternative proposals.

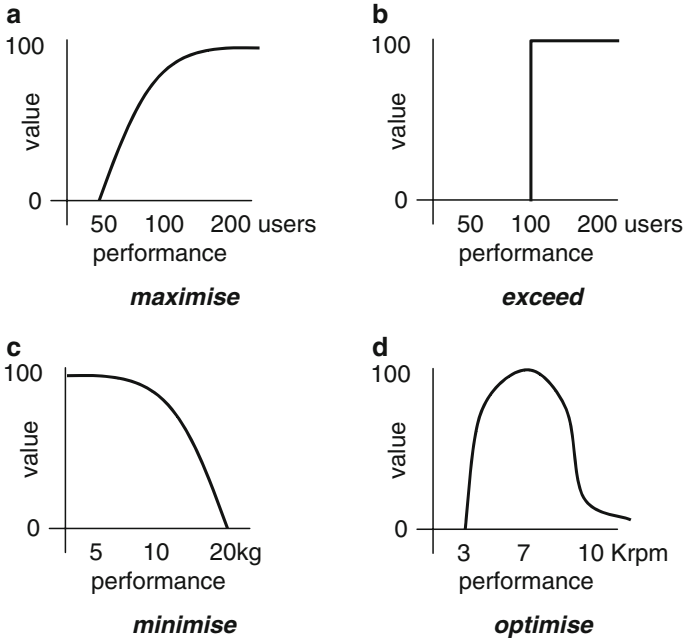

Fig. 4.2 Typical value functions

An attribute can be used to represent a value function as a set of performance/value pairs.

4.8 The Language of Requirements

The use of consistent language makes it easier to identify different kinds of requirements. A simple example of this is the use of "shall" as a key word to indicate the presence of a requirement in the text. Some approaches go so far as to use "shall", "should" and "may" to indicate different priorities of requirement.

The language used will vary depending on the level of requirement being expressed. The principle difference is between stakeholder requirements that lie in the problem domain and system requirements that lie in the solution domain (see Chapter 1, Section 1.9).

As is emphasised in Section 1.9, stakeholder requirements are primarily concerned with capability and constraints on capability. A capability statement should express a (single) capability required by one or more identified stakeholder types (or user groups). The types of stakeholder should be stated in the requirement text.

A typical capability requirement takes the following form:

The <stakeholder type> shall be able to <capability>.

Where there are some aspects of performance or constraint associated solely with the requirement, they may also be stated in the text, for instance giving the form:

The <stakeholder type> shall be able to <capability>
within <performance> of <event>
while <operational condition>.

For example, the following capability requirement has a performance and constraint attached:

The *weapons operator* **shall be able to** *fire a missile*
within *3 seconds* **of** *radar sighting***while** *in severe sea conditions***.**

Less commonly, a single performance attribute is associated with several capabilities. For example, several capabilities may need to be provided with a set time. In practice these capabilities are usually sub-divisions of a high-level capability, to which the performance attribute should be attached.

It frequently occurs, however, that constraints have to be expressed separately from the capabilities, either because they apply to the whole system, or because they apply to diverse capabilities. Generally, constraints in stakeholder requirements are based either on minimum acceptable performance or are derived from the need to interact with external systems (including legal and social systems).

A typical constraint requirement takes the following form:

The <stakeholder> shall not be placed
in breach of <applicable law>.

E.g. **The** *ambulance driver* **shall not be placed**
in breach of *national road regulations.*

Since they lie in the solution domain, the language of systems requirements is a little different. Here the focus is on function and constraints on the system. The language depends on the kinds of constraint or performance associated with the requirement. Here is an example of a function with a capacity performance:

The <system> shall <function>
not less than <quantity> <object> while <operational condition>.

E.g. **The** *communications system* **shall** *sustain telephone contact*
with not less than *10 callers***while** *in the absence of external power*.

Here is another that expresses a periodicity constraint:

The <system> shall <function> <object>
every <performance> <units>.

E.g. **The** *coffee machine* **shall** *produce a hot drink*
every *10 seconds*.

Further discussion of this topic can be found in the following section.

4.9 Requirement Boilerplates

The language of requirements in Section 4.8 was expressed in terms of boilerplates. This section extends this concept, and applies it to the collection and expression of constraint requirements.

Using boilerplates such as the examples in Section 4.8 is a good way of standardising the language used for requirements. A palette of boilerplates can be collected and classified as different ways of expressing certain kinds of requirement. As an organisation gains experience, the palette can be expanded and reused from project to project.

Expressing a requirement through a boilerplate now becomes a process of

- Selecting the most appropriate boilerplate from the palette
- Providing data to complete the placeholders

The requirement can refer to a single document-wide instance of the boilerplate, and placeholders can actually be collected separately as attributes of the requirement. This is illustrated in Fig. 4.3.

From this information, the textual form of the requirement can be generated when needed. Separating the template has the following advantages:

Global changes in style can be effected: To change the ways certain requirements are expressed, only the centrally-held boilerplate needs to be edited.

System information can be processed more easily: Collecting, for instance, all the "<operational condition>" placeholders into a separate attribute allows for easy sorting and filtering on operational conditions.

Confidential information can be protected: In contexts where requirements contain classified or secret information, boilerplates can be used to separate out just those parts of each statement that need to be protected.

This last point merits some elaboration. In military or commercially sensitive projects, there is a need to restrict the availability of some information, but not all. Quite often, a single statement of requirement will contain a mixture of information classified at

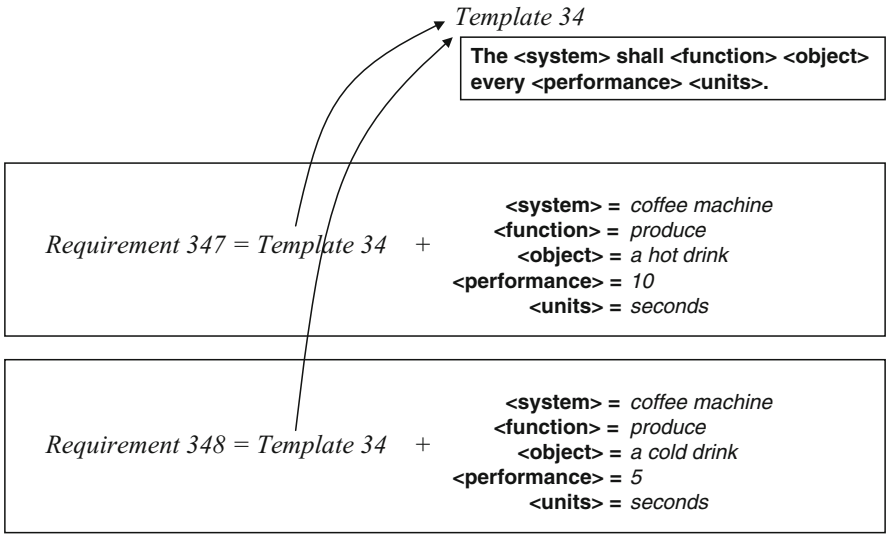

Fig. 4.3 Global templates

various levels. For instance, it is obvious that the ship is going to fire missiles; what is classified is the performance associated with that capability: the state of readiness, the frequency, and the range, etc. Rather than having to hide the whole statement because some of the elements are confidential, boilerplates permit the statement to be visible without some of its more sensitive attributes. Indeed, different readers may be able to see different sets of attributes.

Since there are such a wide variety of constraints, these tend to be the most difficult to express, and this is where boilerplates can help the most. Here is an approach to capturing constraint requirements:

1. Collect all capability requirements first.
2. Construct a list of all the different kinds of constraint that may need to be expressed. If this list is based on past experience of the same kind of system, then boilerplates should exist for each kind. Otherwise suitable boiler-plates may have to be defined.
3. For each capability, consider each kind of constraint, and determine whether a constraint needs to be captured. A large table could be used for this; in each cell, indicate where constraints exist by entering the appropriate sub-ordinate clauses to the requirement; where no constraint is necessary, enter "N/A" in the appropriate cell.
4. Select the boilerplate that best matches the constraint to be expressed, and instantiate it.
5. The process is finished when every "cell" has been considered.

This process answers two frequently asked questions:

- How do I express constraint requirements? (Use boilerplates.)
- How do I know when all constraints have been collected? (Use this systematic coverage approach.)

Table 4.4 shows some examples of boilerplates classified by type of constraint. Note that there may be several ways of expressing similarly classified constraints,

Table 4.4 Example boilerplates for constraint requirements

Type of Constraint	Boiler-Plate
Performance/capability	The <system> shall be able to <function> <object> **not less than <performance> times per <units>.**
Performance/capability	The <system> shall be able to <function> <object> **of type <qualification> within <performance> <units>.**
Performance/capacity	The <system> shall be able to <function> **Not less than <quantity>** <object>
Performance/timeliness	The <system> shall be able to <function> <object> **within <performance> <units> from <event>.**
Performance/periodicity	The <system> shall be able to <function> **not less than <quantity>** <object> **within <performance> <units>.**
Interoperability/capacity	The <system> shall be able to <function> <object> **composed of not less than <performance> <units> with <external entity>.**
Sustainability/periodicity	The <system> shall be able to <function> <object> **for <performance> <units> every <performance> <units>.**
Environmental/operability	The <system> shall be able to <function> <object> **while <operational condition>.**

and that constraints may have a compound classification. Only those parts of the boilerplate that are in bold font are actually relevant to the constraint.

4.10 Granularity of Requirements

The use of requirements boilerplates encourages the practice of placing some constraints and performance statements as sub-clauses of capability or functional requirements. In some cases, it may be desirable to create traceability to and from just those sub-clauses.

This raises the question of granularity of information. How far do we "split the atom" in requirements management?

Statements of requirements can be decomposed into sub-clauses, as long as tool support ensures that clauses are always visible in context. One scheme is to extend the requirements hierarchy to make the sub-clauses children of the main requirement, as shown in Fig. 4.4. Whereas the main requirement is readable (and traceable) on its own, the sub-clauses, albeit separately referenceable for tracing purposes, make sense only in the context of their "parent" statement.

Traceability can now reference a specific sub-clause, but the clause should only ever be cited with the context of its ancestor statements. For instance, the traceable statements that can be cited from Fig. 4.4, with context in italics, are

The communications system shall sustain telephone contact.

- *The communications system shall sustain telephone contact* with not less than 10 callers.
- *The communications system shall sustain telephone contact* while in the absence of external power.

There may be several ways of organising the hierarchy of clauses. Suppose, for instance, that there are multiple capabilities required "in the absence of external power". Then the arrangement may be as in Fig. 4.5.

Now the traceable statements that can be cited are:

- *While in the absence of external power*, the communications system shall sustain telephone contact.
- *While in the absence of external power, the communications system shall sustain telephone contact* with not less than ten callers.
- *While in the absence of external power, the communications system shall sustain radio contact* with not less than 15 ambulance drivers.

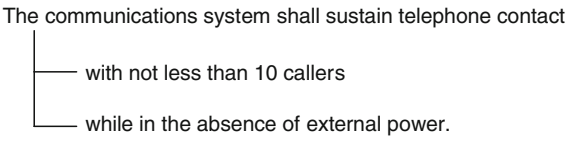

Fig. 4.4 Performance and constraints as sub-clauses

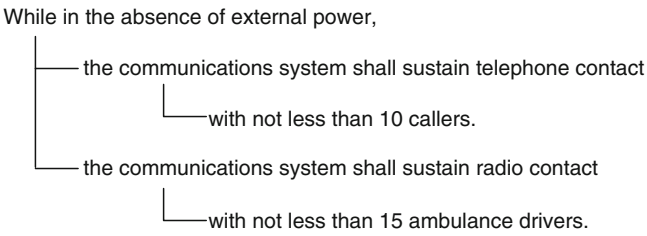

Fig. 4.5 Alternative arrangement of sub-clauses

Indeed, as a general principle, requirements could be organised in such a way that the set of ancestor objects provide the complete context for each statement, including section and sub-section headings.

4.11 Criteria for Writing Requirements Statements

Apart from the language aspects, there are certain criteria that every statement of requirement should meet. These are summarised as follows:

Atomic: each statement carries a single traceable element.
Unique: each statement can be uniquely identified.
Feasible: technically possible within cost and schedule.
Legal: legally possible.
Clear: each statement is clearly understandable.
Precise: each statement is precise and concise.
Verifiable: each statement is verifiable, and it is known how.
Abstract: does not impose a solution of design specific to the layer below.

In addition, there are other criteria that apply to the set of requirements as a whole:

Complete: all requirements are present.
Consistent: no two requirements are in conflict.
Non-redundant: each requirement is expressed once.
Modular: requirements statements that belong together are close to one another.
Structured: there is a clear structure to the requirements document.
Satisfied: the appropriate degree of traceability coverage has been achieved.
Qualified: the appropriate degree of traceability coverage has been achieved.

Two "nightmare" examples of actual requirements are given below.

1. The system shall perform at the maximum rating at all times except that in emergencies it shall be capable of providing up to 125% rating unless the emergency condition continues for more than 15 min in which case the rating shall be reduced to 105% but in the event that only 95% can be achieved then the system shall

activate a reduced rating exception and shall maintain the rating within 10% of the stated values for a minimum of 30 min.
2. The system shall provide general word processing facilities which shall be easy to use by untrained staff and shall run on a thin Ethernet Local Area Network wired into the overhead ducting with integrated interface cards housed in each system together with additional memory if that should be necessary.

Some classic problems are present in these examples. The following pitfalls should be avoided:

- *Avoid rambling*: conciseness is a virtue; it doesn't have to read like a novel
- *Avoid let-out clauses*: such as "if that should be necessary"; they render the requirements useless.
- *Avoid putting more than more requirement in a paragraph*: often indicated by the presence of the word "and".
- *Avoid speculation.*
- *Avoid vague words*: usually, generally, often, normally, typically.
- *Avoid vague terms*: user friendly, versatile, flexible.
- *Avoid wishful thinking*: 100% reliable, please all users, safe, run on all platforms, never fail, handle all unexpected failures, upgradeable to all future situations.

An analysis of the first example above admits that there could be 12 requirements present. A better approach would be to identify clearly the four different operational modes of the aircraft: normal, emergency, emergency more than 15 min, and reduced rating exception, and express a separate requirement for each.

Note the let-out clause in the second example. It is not clear what the scope of the clause is. One interpretation is "The system shall provide general word processing facilities … if that should be necessary." Well is it required, or not?

4.12 Summary

One of the hardest things to do in requirements is to get started. It is important to have an approach, but above all it is important to start writing down the requirements from day 1 and show them to others for comment. The following list is intended as a safe way to proceed:

- Define an outline structure at the outset, preferably hierarchical, and improve it as you go.
- Write down requirements as soon as possible, even if they are imperfect.
- Determine in advance what attributes will be used to classify and elaborate the textual statement.
- Produce an initial version rapidly to stimulate immediate feedback.
- Perfect the requirements as you go, removing repetition, unwarranted design, inconsistency.

- Brainstorm and hold informal reviews continually, with rapid turn-around of versions.
- Exposure to users is much better than analysis by ‘experts’.

The rules to follow when writing requirements are as follows:

- Use simple direct language
- Write testable requirements
- Use defined and agreed terminology
- Write *one* requirement at a time