IBM® Rational® DOORS

Get It Right The First Time

Get It Right the First Time: Writing Better Requirements

Before using this information, be sure to read the general information under the "Appendix: Notices" chapter on page 57.

Table of contents

1 About this Manual

Requirements engineering is a challenge. It involves engineering knowledge, skill in dealing with people, and experience in assessing business realities such as competition and political pressures.

The good news is that many of the traps, which ensnare the unwary, can be avoided by applying the simple guidelines listed here for writing better requirements. Projects that start out well, with a clear and accurate knowledge of the goals and needs of its users, have the best possible chance of success.

Requirements are the key to project success. This book is designed to help both developers and users write better requirements. Because it is about the practical technique of writing requirements, it should be useful in many different kinds of system and software projects. This book enables engineers who have been on a requirements tool course to write requirements well enough for a successful system.

This book specifically focuses on how to obtain better requirements. Together, the book and the associated course, concentrate on the task of actually gathering, writing, and organizing requirements.

The message of this book can be stated in a sentence:

> *Understand and agree upon what users want before attempting to create solutions.*

Finding out what is needed instead of rushing into presumed solutions is the key to every aspect of system development. Most technical problems can be solved, given determination, patience, a skilled team—and a well-defined problem to solve.

2 How to write better requirements

Overview

Defining a good set of requirements is critical for project success. If you can do a good job with requirements, you can save a fortune. So don't feel ashamed about putting effort into ensuring that they are right. The key points of this book are straightforward. Getting a good set of requirements is hardly a technical issue—it involves human interaction and an organized approach. Requirements are the vital communication line between users and developers. System engineering is a human issue, which works only when everyone involved cooperates to solve problems. Projects succeed when they stay focused on results, so that whatever is developed aims only to satisfy the need and deliver only what users want. This is only possible when users' needs are clearly defined, and their problem is fully described in writing.

Here are some tips on getting started:

- Start writing requirements now!
- Produce a document at the soonest, then stimulate immediate feedback.
- Clean up the requirements, removing design and multiple requirements as you go along.
- Define an outline structure first—or if you can't, do it as you go along.
- Brainstorm and hold internal reviews continually.
- Exposure to users and rapid correction is much better than analysis by experts.

Guidelines to keep in mind

1. Keep focused on results during development work, especially within a large project—the objective is to solve a problem experienced by users. Users do not necessarily want the system to have elaborate controls. They do not always need it to look good, to use the latest technologies, or to offer comprehensive documentation compliant with some standard or other. The results that users want are described in the user requirements. The results wanted by users are the developers' goals. They drive the development, and they are the primary test of the system. The system is acceptable if it delivers the desired results.

2. Satisfy the need, and pay continual attention to what it is your users need. Users are human, and will certainly change their views as time goes on. Their working environment is in constant change. Your system, if it is at all successful, will certainly change their way of working still further—so it will change their needs. Be on the lookout for new requirements. These may lead to new business opportunities for your products.
3. Demonstrate success and do not wait for the project to end before showing results. Start on day one by demonstrating that you are in tune with your users' wishes and needs. Early on, create presentations and prototypes to show progress, and to check out your understanding. Divide your project into a series of small and successful steps. That way, there can be no large and unpleasant surprises.
4. Deliver what users want. If the contract is wrong, take steps to agree on changes with your customer. Ensure that the system does whatever it takes to keep deliverables in line with what your users really need. If you help your users to get what they want, they'll do all they can to help you with additional time or resources.

Remember what requirements are for

Writing requirements is not an end in itself. It is not merely extra work or an additional activity to ensure checks and balances or to meet a standard. Requirements have a real purpose in the development of any system.

They are essential:

- To show results the users want from the system.
- To show traceability back to sources and the history of changes.
- To show what the organization needs.
- To show what the system must do.
- To form a basis for the design and design optimization.
- To enable a logical approach to change management.
- To partition the work out to contractors.
- To act as a foundation for testing and payment.
- To test the system or any of its parts during development.
- To communicate the basics about the system in non-technical terms to all participants.

Requirements are a human issue

Requirements are the primary means of communication between users and developers. Within a large project, they may easily be the only way users can tell the developers what they want. A requirements document has the force of a contract behind it, but the needs that it expresses come from people.

The people who write requirements have a difficult job. They must understand the requirements, and be able to organize them into a coherent structure. They have to be able to understand solutions, but not to impose them on the designers. Perhaps the most difficult of all, they have to be *good writers*. They have to produce text that is clear, short, precise, and unambiguous—yet also readable and complete.

In summary, to create better requirements:

- Find out what users want.
- Help organize their needs into a clear document structure.
- Fill the structure with neatly sorted requirements.
- Check it out with users.
- Have it formally reviewed.
- Ensure that the solution stays in line with the requirements as they evolve.

3 Definition and structure of a requirement

A requirement shall be a complete sentence

A single, individual (atomic) requirement must be a sentence. Single words, phrases, and collections of acronyms and abbreviations do not make a requirement. For a requirement to be understood (the first criterion), it must be a complete, correct sentence.

An example of this is as follows:

The reader shall be able to understand the requirement.

This requirement has a subject (the reader) and a predicate (understand the requirement). The phrase *shall be able to* joins the two together to form a complete requirement. Each atomic requirement written must be in this form, or it will not serve the user or the reader well.

The use of a subject implies a user, owner, system under discussion, or design entity to which the requirement will be related.

The predicate should be an action phrase or expression of something that must get done for, by, with, or to the subject.

Shall, will, and must

Consistent use of the verb *to be* solidifies the link between subject and predicate. In requirements language, however, *shall* has a very significant meaning. It signals that this sentence is a requirement and, which must be followed or adhered to.

Other shall words include *will* and *must* to signify a mandatory condition surrounding the requirement.

For example:

The reader must comply with this documentation convention.

This statement relates to what is acceptable to the user insofar as compliance to this requirement.

Less restrictive words are often used to denote non-mandatory or optional requirements.

For example:

The document should take no longer than one hour to read.

Of course, if the document is not readable in one hour, the overall system might still be acceptable. One criterion is merely *desirable* or *optional* and is so written with *should*, *may*, or words like *it is desired that*.

Anatomy of a good requirement

To some extent, each requirement can be analyzed and broken down to perform basic checks. Each user requirement should have:

- a user type who benefits from the requirement
- a defined, desirable state for the user to reach
- a person responsible for implementing the requirement
- metrics or some mechanism to allow a test to be written against the requirement

The requirement,

> *The order entry clerk shall be able to complete ten customer orders in less than an hour.*

can be broken up into component parts. The *order entry clerk* is the user type. The *state* that the order entry clerk reaches is to complete ten customer orders. The requirement is clearly measurable and even shows some performance criteria in completing ten orders per hour. The challenge is to seek out the user type, end result, and success measure in every requirement.

4 Criteria for a good requirement

Good requirements should all follow the same set of criteria no matter the area or description being written about. Check that each requirement is valid against this list. You will find that quite soon you do this automatically whenever you see a requirement.

For each requirement ask the question:

- Is it correct? (asks for something possible, do-able, legal)
- Is it complete? (a complete sentence, as discussed above)
- Is it clear? (unambiguous and not confusing)
- Is it consistent? (not in conflict with other requirements)
- Is it verifiable? (can we determine that the system meets the requirement?)
- Is it traceable? (uniquely identified and can be tracked)
- Is it feasible? (accomplished within cost and schedule)

Other criteria to strive for:

- Is the requirement modular and able to standalone? (non complex, non run-on requirements with few *and/or* references and few external references)
- Does it have a reasonable priority?
- Is it genuinely a user or system requirement? (not a design constraint)
- Is its source shown?

Check the requirements as a set

Next, check the set of requirements as a whole. You need to assess that all the requirements together are realistic, complete, consistent, and concise. These are much tougher questions than those on individual requirements. The only way to be sure they are adequately addressed is for you and the users to have a clear picture in your minds of what the problem is, and for the user requirement structure to reflect that picture accurately. Then you can see when you check over the requirements that they are in fact complete.

Questions of consistency and realism make you think about what each requirement implies, and whether that requirement can be fulfilled in the real world, given all the other requirements. As we shall see, good document structure is vital.

Guidelines for writing good requirements

Provided below are some simple guidelines to follow in writing any requirement. For consistency, the example of an aircraft is used throughout.

1. Define one requirement at a time.

   The pilot shall be able to control the aircraft's angle of climb with one hand.

   The pilot shall be able to feel the angle of climb from the climb control.

2. Avoid conjunctions (and, or) that make multiple requirements.

   The navigator shall be able to view the aircraft's position relative to the route's radio beacons. The navigator shall be able to view the aircraft's position estimated by inertial guidance.

3. Avoid let-out clauses (unless, except, if necessary, but).

   Each cabin crew member shall be provided a rear-facing seat.

4. Use simple direct sentences.

   The pilot shall be able to view the airspeed.

5. Use a limited (500 word) vocabulary (especially if your audience is international).

   The airline shall be able to change the aircraft from business to holiday charter use in less than 12 hours.

   There is no need to use words like *reconfigured*.

6. Identify the type of user who wants each requirement.

   The navigator shall be able to...

7. Focus on stating what result is to be provided for that type of user.

   ...view storm clouds by radar...

8. Define verifiable criteria.

   ...at least 100 miles ahead.

5 Writing user and system requirements

User and system requirements differ in what they define and how they define it. User requirements define the results and qualities the user wants; system requirements define what the system must do to achieve this. User requirements are owned by the users, whereas system requirements are owned by the developers.

Writing the user requirements

The main task of a user requirement is to help users express their needs in writing. For an engineer with a technical or software bias, getting into your users' way of thinking is difficult. Typically, technical people come from a range of different professions, but they usually have their own language and concepts. Requirements are hard to formulate exactly and you may need several revisions to get them just right. Start with a draft and sharpen up the wording later. Initially, just aim to get the basic intention down on paper.

What is a user?

The first step is to define the different user types. Each group of users has a unique set of requirements for the system. The first thing to realize, therefore, is that what you hear depends on whom you ask.

A user is someone involved in using the system when it is actually working. Using is not restricted to only mean operating a system. Anyone involved in maintaining, interacting with, involved in the approval cycle or even checking safety are all users, but a safety inspector is certainly not a system operator.

Examples of users are:

- An order entry clerk
- A shipping and handling clerk
- A salesperson
- A department supervisor responsible for order approval and of course the customer who places the order

System requirements

System requirements define what a future system must do. In this sense they differ from user requirements because they talk about the end product, not just the results that are needed. The system requirements must define how the system has to interface with the systems around it.

System requirements provide an abstract model that shows that they solve every part of the problem, so they must all trace back to the user requirements, which made them necessary.

Systems for businesses of all kinds have to work within limits of safety, performance, reliability, or whatever else is important to the business. Constraints are just as important as actual system functions. What it must not do is as important as what it must do.

Writing system requirements

You can look at system requirements in much the same way as you do user requirements—they say in writing what a system must do to meet the needs of its users.

The big difference is that the system requirements define the system in abstract, sometimes dynamic terms. The user requirements just look at the problem; the system requirements look at the solution and commit to specific technologies while still leaving space for designers to work. System requirements define the system without going into the detail of design. Eventually the designers will come up with just one particular solution, which is their job.

Your job in specifying the system is to make clear to the designers whether any particular solution would be acceptable. To start designing unintentionally is all too easy because it is more comfortable to be definite than to leave choices open.

Just as you make each user requirement begin with a kind of user who wanted that particular result, so you make each system requirement refer to a specific function.

For example:

The order entry system shall provide the ship date of ordered items.

It is then easy for the order entry designers to select only their requirements to work on; not distracted by requirements of other subsystems.

Another distinction of system requirements is that they are not the start of your project. You already have some user requirements, which state what is wanted. You need to show that the system requirements completely satisfy the user requirements. To do this, trace between the two documents so that it is clear where each item comes from, and that it is sufficient for its purpose.

The table below distinguishes between user and system requirements.

User Requirements	**System Requirements**
Description of the problem	Abstract solution
In user language	In developer language
Organized by goals	Organized by functions in a hierarchy or by objects
Subject: a type of user	Subject: the system or subsystem
Defines what the user gets	Defines what the system does
Owned by users	Owned by developers

Making requirements testable

During the requirements work, you need to plan to make the system testable. The system requirements say what the system must do, but these need to be combined with test criteria so that anyone testing the system knows exactly how to carry out each test. For example, the thrust of a jet engine depends on the air temperature and pressure, so the test of that system requirement is not completely specified without these conditions. The details of the tests can be worked out while the design is being made, but the overall plan for the tests needs to be in place first—so it should be ready with the system requirements.

6 Sources of user requirements

Good requirements start with good sources. Finding those quality sources is an important task and fortunately, one that takes little in the way of resources. Time is what is most needed to meld them together to get it right the first time. This task is easiest when you start early and use good people.

So, to get the requirements down on paper, you have to do the following:

- interview users
- visit where they work
- look at other systems
- study suggestions and problem reports
- talk to support people
- study improvements
- look at unintended uses
- assemble prototypes
- arrange workshops and meetings

The best idea is to get the requirements down quickly and then to encourage the users to correct and improve them. Put in those corrections, and repeat the cycle. Do it now, keep it small, correct it at once. Start off with the best structure you can devise, but expect to keep on correcting it throughout the process.

Interview users

Face-to-face contact with users through individual interviewing is the primary source of requirements and an important way you gather and validate them. Remember that it is not the only possible technique, and that interviews can be run in many different styles. Develop a repertoire of styles to suit different situations. Unless you are a user yourself, you will need to make an effort to understand and experience the user's problem to describe it successfully.

Working in the environment

Experience the work of the users for yourself. Working with users helps you understand problems that have resisted previous solutions. Familiar systems developed in this way inevitably include tools for programmers, such as interactive editors and compilers, as the developers naturally have both the expertise in the subject area, and the desire to solve their own problems. It would be good to see the same dedication devoted to solving problems in other areas too. Where the work cannot easily be experienced in this way, it may still be possible to do a bit more than just sit quietly and observe. Users can give you a commentary on what they are doing, what the problems are, and what they would like to have to make the work easier.

Study analogous or existing systems

The starting point for many projects often consist of a similar or existing system. Sometimes comparable products and systems contain working versions of good ideas for solving user problems. You can save the time lost in *reinventing the wheel* by looking at systems already on the market—whether installed at the user's site, or products made by rival organizations. Even if they are trying to solve slightly different problems, they often give valuable clues as to what you should be doing.

Listen when a customer asks why a product couldn't do something that the customer wanted and keep a list of these suggestions. Later, use it to start discussions with other users. You should be able to obtain some requirements directly. If not, capture and store suggestions for future use.

You can describe to users some features of other products, explaining that the system is in another area but contains an interesting concept, ask whether it would help them. Sometimes these systems are described in documents—either a contract from another organization or a report written for management. Often these documents were never intended as formal requirements, and were written to communicate a *stream-of-consciousness*. Define a process of going from a disorganized to an organized information state.

Such a process might involve the following activities:

- Read the document from end to end (several times) to comprehend what the customer wants and what actually has been written.
- Classify all of the types of information in the document. (user, system requirements, design elements, plans, background material, irrelevant detail)
- Mark up the original text to separate out such requirements.
- Find a good structure for each of the different types of information—a scenario for the user requirements, functional breakdown for the system requirements, architecture for the design.
- Organize the information to show gaps and overlaps. You should feel free to add missing elements—but confirm these decisions with the users.

- Create traceability links between these information elements to show the designers exactly what is wanted.
- Convince the customer to accept the new information as the basis for the contract.

Examine suggestions and problem reports

Requirements can come from change suggestions and user problems. A direct road to finding requirements is to look at suggestions and problems as first described. Most organizations have a form for reporting system problems or software defects. You can ask to look through the reports and there will probably be many. Sort them into groups so you can identify the key areas that are troubling users. Ask users some questions about these areas to clarify the users actual needs.

Requirements from the help desk and support teams

All large sales organizations have a help desk, which keeps a log of problems and fixes, and some support engineers who do the fixing. Many organizations have similar facilities to support their own operations. Talking to the help desk staff and the support engineers may give you good leads into the requirements, and save you time. Talk to the training team and installation teams on what users find difficult.

Improvements made by users

This is an excellent source of requirements. Users of a standard company spreadsheet may have added a few fields, or related different sheets together, or drawn a graph, which exactly meets their individual needs. You need only ask, *what is that for?*, *Why did you add that?* to get to the heart of the actual requirement. This applies also to hardware and non-computer devices. For example, a lathe operator may have manufactured a special clamp, or an arm that prevents movement of the tool beyond a certain point. Any such modification points to something wrong with the existing product—and is therefore a valid new requirement for the new version.

Unintended uses

They are an immensely creative way to get new ideas and innovations. An observant product manager noticed that an engineer was staying in the office late to use an advanced computer-aided design system to design a new kitchen layout for his home. Inexpensive commercial products, with names like *Kitchen Designer*, are now widely available for home use.

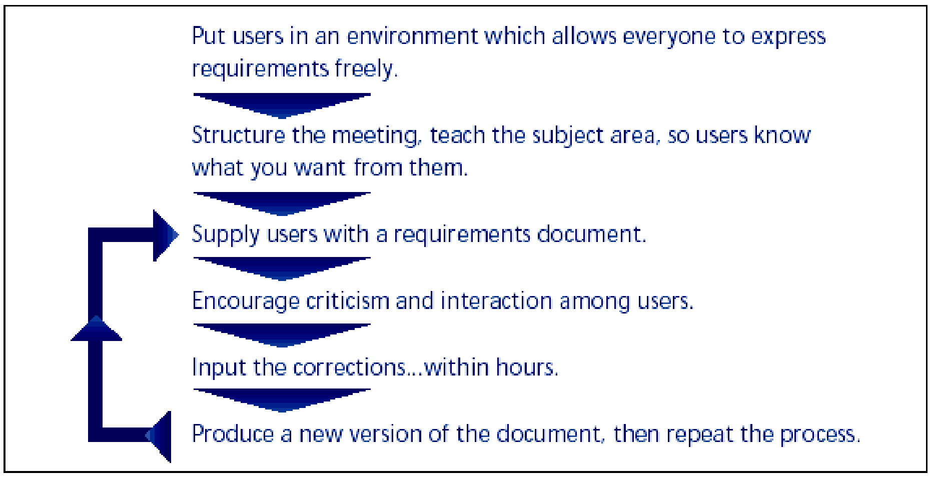
Workshops

They can rapidly pull together a good set of requirements. In two to five days, you can create a set of requirements and then review and improve them. If everyone in a workshop tries to estimate the cost and value of each requirement, the document becomes much more cost-effective.

Workshops are quicker and better at discovering requirements than other techniques such as sending out questionnaires. You are bringing the right collection of people together, and getting them to correct and improve on their requirements document.

A workshop is inherently expensive because of the number of people involved, but it does save a large amount of time. If you can define the product right the first time and cut three months off the requirements phase, the savings could be enormous. The workshop has to be thoroughly organized to take advantage of people's time.

Choose a quiet location for the workshop so that people are not disturbed by day-to-day business. Mobile phones should be discouraged; arrange to take messages externally. Take advantage of informal interactions by choosing a site so that people don't go home at night or go out separately. The example below shows the logic of a requirements workshop.


Put users in an environment which allows everyone to express requirements freely.
Structure the meeting, teach the subject area, so users know what you want from them.
Supply users with a requirements document.
Encourage criticism and interaction among users.
Input the corrections...within hours.
Produce a new version of the document, then repeat the process.

Prototypes

They allow us to immediately see some aspects of the system. Showing users a simple prototype can provoke users into giving good requirements or changing their mind about existing requirements. The techniques described here help you gather ideas for requirements. Prototypes and models are an excellent way of presenting ideas to users. They can illustrate how an approach might work, or give users a glimpse of what they might be able to do. More requirements are likely to emerge when users see what you are suggesting.

A presentation can use a sequence of slides, storyboard, an artist's impression, or even an animation to give users a vision of the possibilities. When prototyping software, make a mock-up of the user interface screens, emphasizing that there is no code and that the system has not been designed or even specified yet (there are dangers here for the unwary).

This prototyping aims to get users to express (missing) requirements. You are not trying to sell users an idea or product, you are finding out what they actually want. Seeing a prototype, which invariably is *wrong* in some ways and right in others, is a powerful stimulus to users to start saying what they want. They may point out plenty of problems with the prototype! This is excellent, as each problem leads to a new requirement.

7 Organizing requirements

You need structure as well as text

A medium-sized project typically demands hundreds of requirements. Readers can only understand these fully when the requirements document is organized logically. The best structure for user requirements follows the natural patterns of use, structured like a scenario or use case.

Everyday language is the only medium that users and developers share. Everyone can immediately understand requirements so written; therefore you might think writing in plain language would be the ideal way to define user needs. But simple text isn't good at showing how different user needs fit together. After all, you want to make a system, not a mass of unrelated functions. So, you need to make a structure that will organize the requirements.

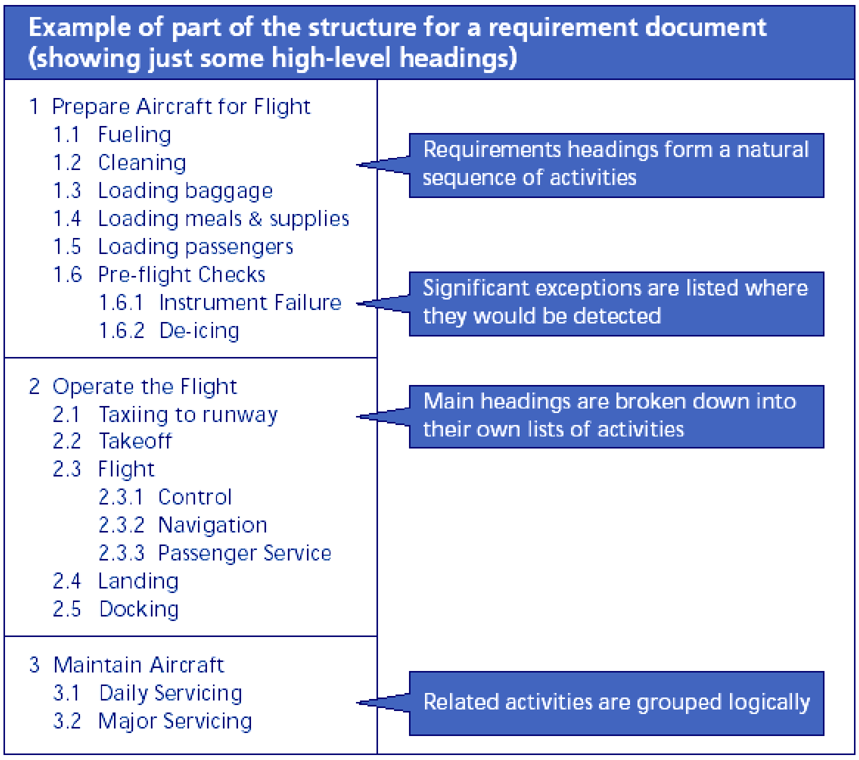
The structure must give readers insight into the text. A good structure shows the requirements at different levels of detail and allows readers to focus on one section at a time. The users have to understand and dominate the user requirements, whoever wrote them. A combination of textual requirements and a scenario-like structure of section headings is very effective. This can be supplemented if you use simple diagrams.

Diagrams must be easy for users to understand. Most kinds of engineering diagrams are not suitable for non-technical users. Dataflow analysis creates hundreds of diagrams, so users feel lost, and they give up. Flowcharts can get big, losing users in a mass of detail.

The goal is to show users just one simple structure, but also allow for any amount of detail. To do this, arrange what users need under chapter headings. Users can then see the whole pattern before diving into the details. You probably need three levels of headings to organize the requirements. Try not to use many more than that. Your structure should explain itself.

Organizing requirements in practice

The following example shows you one way to organize your document.



Example of part of the structure for a requirement document (showing just some high-level headings)

1 Prepare Aircraft for Flight
  1.1 Fueling
  1.2 Cleaning
  1.3 Loading baggage
  1.4 Loading meals & supplies
  1.5 Loading passengers
  1.6 Pre-flight Checks
    1.6.1 Instrument Failure
    1.6.2 De-icing

2 Operate the Flight
  2.1 Taxiing to runway
  2.2 Takeoff
  2.3 Flight
    2.3.1 Control
    2.3.2 Navigation
    2.3.3 Passenger Service
  2.4 Landing
  2.5 Docking

3 Maintain Aircraft
  3.1 Daily Servicing
  3.2 Major Servicing

Requirements headings form a natural sequence of activities
Significant exceptions are listed where they would be detected
Main headings are broken down into their own lists of activities
Related activities are grouped logically

Work out a complete scenario-like heading structure (top two levels, as illustrated) for the capabilities chapter of the user requirements document for a passenger aircraft.

Defining the scope

Projects only succeed when they know what they must accomplish. It is vital to define exactly what your system does and does not have to cover, from the start. System scope is like a hole, perhaps defined by what it is not, rather than what it is.

Agree on exactly what to include

A clear view about what to include is critical. All too often there is confusion about whether something is in or out of a system. For example, users may have legitimate requirements that are impossible to meet in the time available or too expensive for the customer's pocket. So the system's scope has to be cut down to ensure success.

The scope of any system is defined by negotiation between the customer, who states what is needed from a business perspective, and the developer, who says what is practical. Technical users always want more than the customer is willing to pay for. Make sure you know who has the final say on system scope.

Identify priorities

The best approach to scoping is for the customer to state up-front what is needed, even before the requirements are collected. Of course, the customer must never stop users from asking for what they want, even if it is slightly out-of-scope. That does not mean that those requirements will necessarily be implemented. Keep them, but mark them as *to be implemented later*—in other words, give them a priority. This is far better than deleting requirements, only to have them re-introduced and re-argued at the next review meeting.

Work out what can be afforded

Sometimes a system has to be limited in scope because a requirement cannot be met with the available budget. The first rule here is not to despair. Developers can often suggest simpler alternatives that will do 80% of what the users wanted.

Once the customer has made clear how much can be afforded, developers and users can sit down with the requirements and work out how to get as much as possible done within the budget. They may well be able to implement several of the *to be implemented later* requirements in a modified form.

It is vital that you agree upon any such compromises in advance.

Putting the requirements in the right place

After collecting user requirements from different sources, you need to sort them out so that they make sense and can be understood when taken as a whole.

The most natural way to make the document do this is to arrange requirements around a basic sequence of goals which deliver what users want, enhanced with extra sequences to cope with exceptions. You'll know you have the structure right when users of any type instantly see how their part fits into the whole thing.

Identify where each requirement fits into the structure and put it there. This task demands that you view the structure as a hierarchy, not just as a list or table. If there is still no appropriate section for a requirement, create one and agree on changes with the users.

8 Requirements—Aids for Analysis

One of the key benefits of having requirements is that you can view and manage the system without having it physically built. This is especially important in software-intensive projects. Two key uses of the body of requirements are to make engineering decisions and to manage the project.

Status information is essential

Up to now, this book has described requirements as if they were just pieces of text, probably single sentences containing the word *shall* and as few other words as possible. This is fine as far as it goes, but it is not the whole story. Each requirement consists of attributes, or status values, as well as text.

Here's a checklist of actions to enable people on your project to track the source and status of your requirements. The values attached to each requirement are most easily handled with a requirements tool.

- Record the source or who suggested the requirement (also record when it was created and any rationale along with the original text).
- Record how far the requirement is towards being accepted (proposed, reviewed, accepted, rejected, to be modified).
- Record how urgently the requirement is wanted (mandatory, useful, optional, luxury).
- Record the requirement's priority in the development of any future system (date, version, or release number).
- Identify how the requirement will be verified (test, simulation, inspection, analysis).
- Record any other constraint or related information (safety, performance, or reliability attributes).
- Record any questions against the requirement (comments, action items or recommended changes).

Analysis made simple

As you can see, a wealth of information can be captured in addition to just the narrative requirement. Once this information is captured, questions concerning the status of the project can be answered.

- Show all the high priority requirements from the pilot on the flight control sub-system.

  Priority = High

  User = Pilot

  Subsystem = Flight Control

- Show those mandatory requirements in this release or version of the product.

  Urgency = Mandatory

  Release = 1.0

- Show all the requirements the maintainers proposed that could affect safety and performance.

  User = Maintainer

  Requirement Status = Proposed

  Related Constraints = Safety & Performance

There is no limit to the power of this kind of structured analysis. Without requirements, the project may still ask some of these questions but the answers might not be based on the same data, or yield consistent results. A word of warning as well, be careful of using more attributes than you have time to maintain. The result is often out-of-date data.

9 Reviews

Reviews are meant to discover problems early enough to solve them quickly and cheaply. Reviews can be informal—showing draft documents to your colleagues, or correcting an outline structure. These informal reviews are excellent for getting the right structure and removing obvious mistakes.

Formal reviews are more wide-ranging and expensive. They start with careful preparation, so that comments are organized in time for the formal meeting. The meeting itself produces decisions on all review items. After the meeting, the review actions must be pursued to completion.

The review process

The review process involves three main steps:

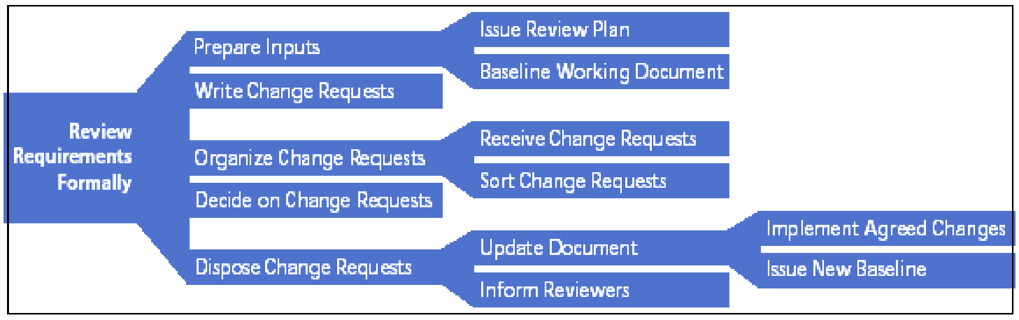
- collecting review comments from all stakeholders
- deciding what to do about the comments, usually by agreeing to changes to the requirements
- making the agreed upon changes to the document

Only the middle stage happens in the review meeting. Getting all the participants or stakeholders together in a meeting is expensive, so perform as much work as possible beforehand.

The stakeholders should review the document on their own time before the meeting, writing comments and sending them in to the review organizer. The organizer has an intense job, collecting and sorting all the comments to ensure the meeting runs smoothly.

The purpose of the meeting is to make decisions on the review suggestions, which are available in advance to everyone.

The job of editing the requirements document or taking any other action (such as finding out extra information) is done as soon as possible after the meeting. The following example shows the structure of requirements for holding a review.



The review meeting

The review meeting aims to improve the quality of requirements. In reviewing, focus the maximum amount of intelligence onto a document to get it as close as possible to telling the developers exactly what users want.

Reviewing needs to be treated as an opportunity to improve the state of the program to get the best results. Different participants have their own intentions; such as to make their own lives as convenient as possible, or to get their own requirements satisfied regardless of the impact on everybody else. This is why reviews must involve everybody with an interest in the problem, and why a *leader* must be in control of the proceedings.

Remember, take the attitude that every suggestion is a gift.

Capturing the suggestions

First, the requirements have to be frozen. Keep an exact record of the version number and date of everything you send out to reviewers, and keep a reference copy in your project library. Make a backup of the review documents on a disk or tape. For small informal developments, reviewers can write over (identical) printed copies of the document and send them back in. But for bigger projects, it is better to suggest changes on a specific form.

Collect all the suggestions, and sort them into the order of the document. Where several people have made the same point, *staple* the suggestions together so that they are handled as one.

Make the necessary changes

After the review, plan to step through the suggestions, deciding whether to accept or reject them. Document all your decisions as you go along. If more work is needed before a decision can be reached, allocate some time for that work. Be sure to address all the decisions and have them implemented in an updated version of the requirements document. Publish minutes that identify the suggestions and decisions, and provide the disposition of the actions.

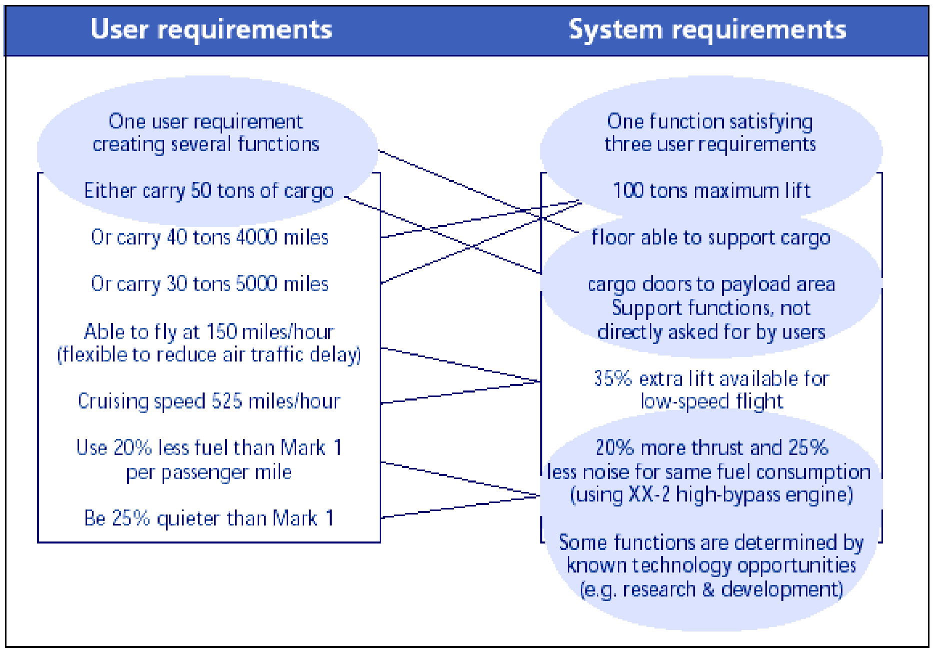
Golden rules for reviewing:

1. Encourage criticism—remember that people are improving the document, not criticizing you. Even the harshest criticism often contains a grain of truth.
2. A few specific people make the best reviewers, time and again. Cultivate them and make sure they have time allocated for the work.
3. It's not over until the corrections have been made and agreed upon. Allocate the available time sensibly so that you cover all suggestions. Only three decision outcomes are allowed: accept, reject, and accept with modifications. Try to make about 30 decisions per hour. You don't have to be nice to be a good reviewer—in fact it might help if you are not!
4. Reviews are great for making small decisions, but useless where a large amount of work is required. Get the structure of the document right before the review meeting, and then all the change suggestions can easily be accommodated.
5. Work hard to minimize inter-relationships and make the requirements easy for people to understand. To review any individual requirement in a document, people have to understand the whole set of requirements, and the inter-relationships between them. 800-page specifications can destroy a project because they are simply impossible to read.
6. Reviews are expensive because they tie up so many people, so aim to get everything right in a single formal review. Small-scale reviews and inspections with your local colleagues can happen anytime. Continue with these until you are sure that passing the formal review will be just a formality.

10 Traceability

Large systems are complicated. Their complexity can only be handled if you divide up the problem into manageable parts. The largest parts are levels—system, subsystem, and so on. Each of these parts has its own requirements, and is run as a separate project or subsystem.

The complexity of large systems


User requirements
System requirements
One user requirement creating several functions
Either carry 50 tons of cargo
Or carry 40 tons 4000 miles
Or carry 30 tons 5000 miles
Able to fly at 150 miles/hour (flexible to reduce air traffic delay)
Cruising speed 525 miles/hour
Use 20% less fuel than Mark 1 per passenger mile
Be 25% quieter than Mark 1
One function satisfying three user requirements
100 tons maximum lift
floor able to support cargo
cargo doors to payload area
Support functions, not directly asked for by users
35% extra lift available for low-speed flight
20% more thrust and 25% less noise for same fuel consumption (using XX-2 high-bypass engine)
Some functions are determined by known technology opportunities (e.g. research & development)

An aircraft maker places thousands of requirements on a new type of plane. The people who make the components and subsystems for the plane—engines, control units, fire detectors, navigation devices, radios, lights—place many thousands more. The airlines who buy the plane have simpler needs: it can fly so far, so fast, carrying so many passengers and so much cargo, in such-and-such comfort, using only this amount of fuel and needing servicing only that often. Finally, the passengers, who are the ultimate users, just want to arrive quickly and safely, and preferably cheaply. The above example shows some of this relationship.

Many relationships exist between all these people and requirements. The engine maker is the *user* of the engine's fire detector and, in a different way, of the engine control unit. The aircraft maker is the *user* of the engine. The airline is the customer for the aircraft. The passengers are, in a different way, the customers for the airline, but they don't buy the planes. Instead, the airline's marketing department asks existing passengers what they want from an airline of the future, and writes a report, reflecting the many statements they get back.

Systems can be critical to business for safety, for public liability, for damage to property, or more directly for collecting (or failing to collect) revenue. A telecommunications company customer billing system brings in all the firm's income. If it goes down for just an hour it will give the customers free calls for that time, and probably millions of dollars' worth, money which can never be recovered.

Requirements allow customers to get what they want, but only if they effectively control developers (or subcontractors). Control means that each item can be traced to its requirements, and tested against them. Traceability is therefore a vital tool in managing system development through requirements.

11 Controlling change

Controlling change is a matter of recognizing the forces of change and being prepared to respond to that change.

Forces of change

Even after a good set of requirements has been created, reviewed, and agreed upon, these requirements will continue to change. This is an inevitable fact of system development. New requirements and design possibilities emerge. Schedules shift; organizations such as regulatory agencies, customers, and suppliers reorganize, merge, are taken over, and go bankrupt; offices are relocated, disrupting your infrastructure and carefully optimized network architecture. New technology may make some requirements easier to implement. A competitor may add a critically important feature that demands a quick response.

Tracking change

As a result of these risks, you constantly need to check for changes in the design and requirements, just to keep up. To do this, you have to trace from each requirement to the design component that satisfies it. You have to check that the design is in fact sufficient to meet the requirement. A single requirement change may affect several design elements. As well, a single design modification can sometimes affect several requirements, so the change relationships can become complicated. A well-run project will have a method of tracking change in place before the change begins to occur. Many organizations couple their requirements change proposal system with their action item or defect tracking system. This is not altogether a bad idea, if it makes tracking and managing change easier and a more natural part of business.

Allow for feedback

Always make sure you allocate enough time for user feedback. After your first attempt you have set up a framework in which users feel comfortable making informal comments on their requirements. Take time and effort to co-operate in a relaxed and open way with your users. Foster an environment that encourages and allows for feedback. Make clear to your team that the users are part of the team. A spirit of co-operation is essential. If users and program participants feel their comments are not listened to, the suggestions and ultimately the support will stop coming and everyone will suffer.

Requirements effort throughout the lifecycle

Some effort is needed throughout the project, because compromise and change are inevitable. Do not be discouraged that requirements are inevitably changed through the course of a project. As it becomes clear what is feasible, and how much requirements will cost, you will have to compromise. At this time, you will need to know how important the requirements are to users.

Other helpful hints on managing change

1. Allow for change—Organize the requirements so that you can cope with the changes and fund efforts to periodically review and update the requirements throughout the project.
2. Allow for users' feelings—Some users may be defensive about giving their opinions, especially if, for instance, they think their jobs may be affected by the system being developed. In that situation, it is essential to gain their trust before trying to start developing a system. Make sure that management, users, and developers share an understanding of what the system will mean for the workforce.
3. Keep the requirements in the forefront—Once the formal reviews are completed and the design is well under way, make sure everyone keeps in mind why we are building the system. Plan to stop and re-review the user and system requirements periodically in design and again in test. Several months or perhaps a year may have passed since these requirements have been written. If the requirements need to be changed, then start that process. If the implementation is not tracking them, it's time to adjust.

A Appendix: Writing pitfalls to avoid

Listed below are some pitfalls to avoid in defining and writing requirements. In some senses they are an inverse of the definition of writing good requirements. In other cases showing examples of what not to do can help explain better.

Avoid ambiguity

Avoidance of ambiguity is one of the subtlest and most difficult issues in writing requirements. Try to write as clearly and explicitly as possible. Remember if this is carried too far, the text becomes dull and unreadable—and therefore cannot be improved by other people. Although this book emphasizes structured written expression of requirements, informal text, diagrams, conversations, and phone calls are excellent ways of removing ambiguity.

Dangerous ambiguities can be caused by the word *for*, and also by many more subtle errors.

Example:

> *The same subsystem shall also be able to generate a visible or audible caution/warning signal for the attention of the co-pilot or navigator.*

Which subsystem? Is the signal to be visible, audible, or both? Is it both caution and warning, just caution, or just warning? Is it for both the co-pilot and the navigator, or just one of them? If just one of them, which one and under what conditions?

Don't make multiple requirements

Requirements which contain conjunctions (words that join sentences together) are dangerous. Problems arise when readers try to puzzle out which part applies, especially if the different clauses seem to conflict, or if the individual parts apply separately.

Dangerous conjunctions include *and*, *or*, *with*, *also.*

Example:

> *The battery low warning lamp shall light up when the voltage drops below 3.6 Volts, and the current workspace or input data shall be saved.*

Never build in let-out or escape clauses

Requirements which contain let-outs or escapes are dangerous. They look as though they are asking for something definite, but at the last moment they back down and allow for other options. Problems arise when the requirements are to be tested, and someone has to decide what (if anything) could prove the requirement was not met.

Dangerous let-outs include: *if*, *when*, *but*, *except*, *unless*, *although*.

Examples:

> *The forward passenger doors shall open automatically when the aircraft has halted, except when the rear ramp is deployed.*
>
> *The fire alarm shall always be sounded when smoke is detected, unless the alarm is being tested or the engineer has suppressed the alarm.*

(This is a truly dangerous requirement!)

Don't ramble

Long rambling sentences, especially when combined with arcane language, references to unreachable documents, and other defects of writing, quickly lead to confusion and error.

Example:

> *Provided that the designated input signals from the specified devices are received in the correct order where the system is able to differentiate the designators, the output signal shall comply with the required framework of section 3.1.5 to indicate the desired input state.*

Refrain from designing the system

Requirements specify the design envelope, and if we supply too much detail we start to design the system. Going too far is tempting for designers, especially when they come to their favorite bits. Danger signs include names of components, materials, software objects/procedures, and database fields.

Example:

> *The antenna shall be capable of receiving FM signals, using a copper core with nylon covering and a waterproof hardened rubber shield.*

Specifying design rather than actual need increases the cost of systems by placing needless constraints on development and manufacture. Often knowing why is much better than knowing what.

Avoid mixing different kinds of requirements

The user requirements form a complete model of what users want. They need to be organized coherently to show gaps and overlaps. The same applies to system requirements—form a complete functional model of the proposed system. A quick road to confusion is to mix up requirements for users, systems, and how the system should be designed, tested, or installed. Danger signs are any references to system, design, testing, or installation.

Example:

> *The user shall be able to view the currently selected channel number which shall be displayed in 14pt Swiss type on an LCD panel tested to Federal Regulation Standard 567-89 and mounted with shockproof rubber washers.*

Do not speculate

Requirements are part of a contract between customer and developer. There is no room for *wish lists*—general terms about things that somebody probably wants.

Danger signs include vagueness about which type of user is speaking, and generalization words: *usually*, *generally*, *often*, *normally*, *typically*.

Example:

> *Users normally require early indication of intrusion into the system.*

Do not play on ambiguous requirements

Some constructions (such as the use of *or* and *unless* in requirements) allow different groups of readers to understand different things from the same wording. Developers could use this technique deliberately, so as to postpone, until too late, any possibility of the customer's asking for what was truly wanted. This is dangerous and could easily backfire.

The only approach that works is for developers to make requirements as clear as possible, and for all stakeholders to co-operate. In the long run, project success is in everybody's interest.

Do not use vague undefinable terms

Many words used informally to indicate system quality are too vague for use in requirements.

Vague terms include: *user-friendly*, *versatile*, *flexible*, *approximately*, *as possible*.

Requirements using these terms are unverifiable because there is no definite test to show whether the system has the indicated property.

Examples:

> *The print dialog shall be versatile and user-friendly.*
>
> *The OK status indicator lamp shall be illuminated as soon as possible after the system self-check is completed.*

Do not express possibilities

Suggestions that are not explicitly stated as requirements are invariably ignored by developers.

Possible options are indicated with terms such as: *may*, *might*, *should*, *ought*, *could*, *perhaps*, *probably*.

Example:

> *The reception subsystem probably ought to be powerful enough to receive a signal inside a steel-framed building.*

Avoid wishful thinking

Engineering is a real-world activity. No system or component is perfect. Wishful thinking means asking for the impossible.

Wishful terms include: *100% reliable*; *Safe*; *Handle all unexpected failures*; *Please all users*; *Run on all platforms*; *Never fail*; *Upgradeable to all future situations.*

Examples:

> *The gearbox shall be 100% safe in normal operation.*
>
> *The network shall handle all unexpected errors without crashing.*

B Appendix: List of possible constraints

Many requirements are not functions, but qualities of behavior that users want. In this way they *constrain* the system within acceptable operational bounds. These constraints are also referred to as non-functional requirements—again, because they do not specify functions of the systems. Examples of the most common and important non-functional requirements include performance, interface, safety, and a group referred to as the *ilities* (reliability, availability, maintainability, portability, and others). Provided below is a list of some constraints and a brief definition of each. Your system may have other constraints resulting for the uniqueness of your problem area. Do not hesitate to define new and specific constraints.

Two general guidelines about constraints:

1. They are best linked to a functional requirement.
2. Group them together in your information model or document to review and evaluate them together.

Performance requirements

Performance requirements typically quantify the operational value of other requirements. Performance requirements are often a numeric value assigned to the requirement, or even more typically have a relationship between two or more requirements. Make sure they are visualized together and that the end-to-end performance constraints apply to the whole system. Examples of performance requirements include calls per hour, response time, flight range, and number of fixes.

Interface requirements

Interface requirements in a User or System requirements document usually means a wholly external companion or co-operating system. In this case an interface requirement should specify only two end points and specify the purpose and what the payload across the interface is.

Safety requirements

Safety requirements define those qualities the system must have to ensure safety of the system. These include rounded edges, grounding, hazardous materials, noise levels, and all the hazards and faults that could threaten safety. Define safety requirements by analyzing the functions, and later the design, of a system to consider how likely each hazard is, what can be done to reduce it to an acceptable level, and what to do if a breach of safety occurs.

Training requirements

Training is often an issue in large systems as a good number of people must be trained, retrained, and perhaps certified. Training requirements should specify the type and location of training (embedded, classroom, individual, and field), desired length of training (hours, days, and weeks) and type of people to be trained. State any pre-requisites or start-of-training criteria the system might place on those trained.

Documentation requirements

Most systems require a large amount of documentation upon system delivery. Types of documents, such as user manuals, installation guides, training material (related to training requirements), and maintenance manuals are some examples. Guidelines on length of documents, reading level, format, and medium (paper, CD, or Web-Based) may also be specified.

Reliability

How long must the system function under normal and abnormal conditions? Minimum times (minutes, hours, days) should be specified as to how long the system should operate before incurring downtime, reboots, or going off-line.

Portability

Defines the desired level of system portability to other platforms, usually in terms of percentage of code to be rewritten in order for the software to run on another system. Portability requirements can also mean requirements for different methods of transportability (rail, air, truck or ship) and any such consideration like fuel level, max wind speed protection, shake, rattle, and roll.

Maintainability

Description of the system's maintenance requirements should be specified. Types and level of maintenance (User, Level 1, Field, preventative, and manufacture) should be defined as well as distribution method for updates, who will perform the maintenance, and any documentation required (related to documentation requirements). Others include time between repairs, time to repair, and any warranty information.

Availability

Length of time the system should be ready for use is in-service, or operational. Often specified as a percentage of time during a given interval. Can also specify or point to what actions to take if the time or percentage drops below that level (call maintenance, issue a report, and so on).

C Appendix: Breaking down requirements

Part of defining a good set of requirements is in breaking a high-level activity or function down into smaller steps or sub-functions. This part of getting it right means the following:

1. Set the main goal.
2. Define the key steps.
3. Decompose or break them down into smaller steps.
4. Relate any links between smaller steps or other requirements.

The goal

First, identify the goal for the whole problem. This is simple if you begin with the word *to*, as this gets you thinking about a single action or mission. For example, the goal for a transportation enterprise is *to get the goods delivered*.

The goal for an accounting office is *to get payment for the company's products*.

Key steps

Next, write down the short sequence of key steps that you must take, to achieve the goal. You may be able to do this directly with the help of some users. Starting at the end (having reached the goal) and working backwards is the easiest approach.

For example:

- To get payment into your company's account, you have to pay in the customer's checks.
- To get the checks, you have to send out invoices to the customers.
- To send the invoices, you obviously have to work out what they owe you.
- To work out ...

You can see why it helps to work backwards a step at a time, because the answer, for each small step, is usually simple.

Organizing the user requirements into a time sequence (or *operational scenario*) works well. For a first cut, the key results needed by users form a single *normal business* sequence: a clear flow from first to last. You can step through the structure like a scenario, and quickly detect gaps and duplication. Later, you will add other sequences, such as abnormal behavior or exception handling. These will build more structure around the first-cut sequence. For example, you can expect emergency actions to branch off from the sequence of normal actions.

Decompose or break down into even smaller steps

Once the high-level steps to achieving the goals are agreed upon, decompose each step into the smaller steps needed to reach its goal, and again check out your understanding with the users.

If progress towards the goal is built up in stages, the final step to reaching the goal is similar to the goal itself. Conversely, sometimes several goals can be worked on simultaneously. For example, to respond to a major incident, three sub-goals can be worked on in parallel, and none of them resemble the goal.

Some steps may involve plenty of work. Calculating the invoiced amount for an important customer may mean keeping a complete database covering orders, previous payments, discounts, state taxes, delivery charges, credit notes, and much more. So you can repeat the process just explained, breaking each step down as if it were the goal.

The steps leading to a goal are usually simple for users to identify. See how much easier it is to understand the problem when it is arranged in order! Structure makes it easy to see what is there—and what is missing.

Make the structure easier to understand by:

- emphasizing the sequences with numbered headings
- choosing larger type for the higher-level headings
- indenting the headings differently from the text

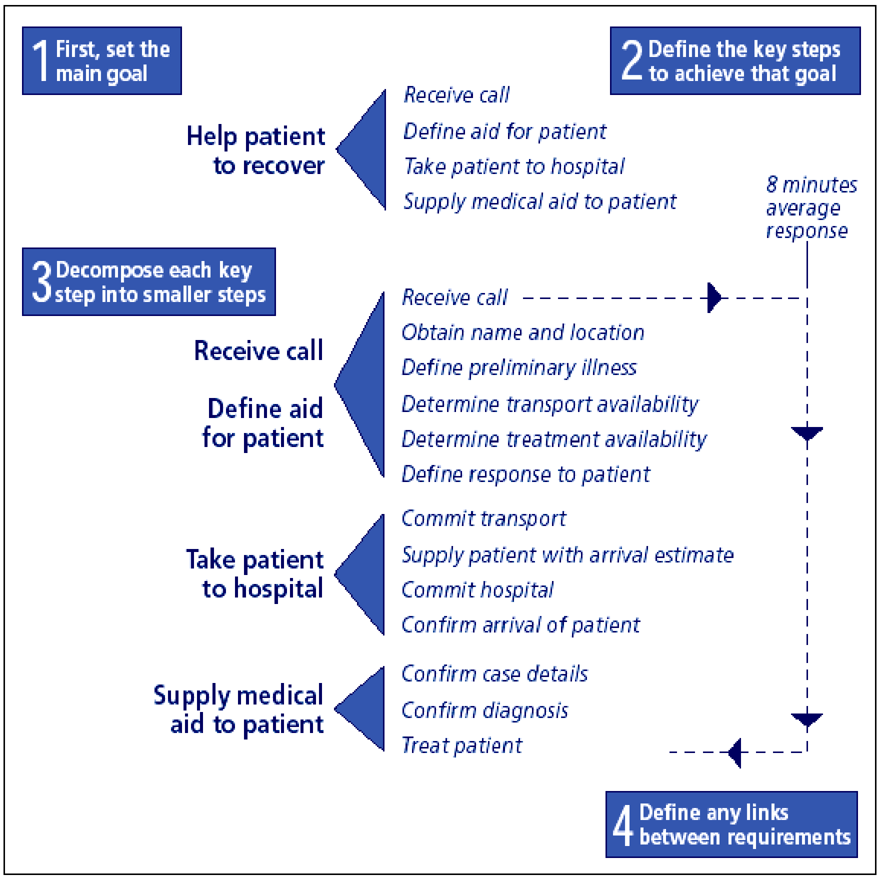
Define any relationships

One of the main by-products of decomposition is the relationship between other requirements as we break them down. Make sure you take time to understand and capture each of these relationships, as they will be extremely valuable later on.

An example

To illustrate this, the following example shows a simple patient aid system. Start with the top-level goal—the end result that the user actually wants from using the system—to help the patient recover. Then define the steps or subclass leading to that goal. Review the set of results that these high-level goals provide:

are they what the users actually want? In the illustration, the results are that the patient's call is received, the aid needed by the patient is defined, the patient is taken to a hospital, and the defined medical aid is supplied to the patient. Together, these results plainly do achieve the top-level goal, which is to help the patient recover. How each of the results is to be achieved is not yet defined; but fortunately, even without that knowledge, users can agree with—or correct your understanding of—the goals. Using this approach, you can therefore be sure at every stage that the requirements are correct.


1 First, set the main goal
2 Define the key steps to achieve that goal
Help patient to recover
Receive call
Define aid for patient
Take patient to hospital
Supply medical aid to patient
8 minutes average response
3 Decompose each key step into smaller steps
Receive call
Receive call
Obtain name and location
Define preliminary illness
Define aid for patient
Determine transport availability
Determine treatment availability
Define response to patient
Take patient to hospital
Commit transport
Supply patient with arrival estimate
Commit hospital
Confirm arrival of patient
Supply medical aid to patient
Confirm case details
Confirm diagnosis
Treat patient
4 Define any links between requirements

To prepare to treat patients in a major incident:

- Summon backup medical staff to hospital.
- Prepare casualty department.
- Summon backup ambulances to incident.

D Appendix: Attributes

Attributes are a very important source of requirements information. Just as every person has attributes (age, hair color, gender), each requirement has a source, a relative importance, and time it was created. Attributes not only extend the meaning of each requirement; if created properly, they can yield significant information about the state of the system. Just as a query can be made on all men with brown hair over age 30 given our human example, queries on the status of requirements can be made such as all high-priority requirements from the customer in the last 30 days.

Examples of attributes

Listed below is a partial list of some common attributes and a brief description of their meaning. Some attributes are best described as a number, date, boolean (true or false) or a text field for entering free format comments. Other attributes can be expressed as lists. For instance, priority type is a list of high, medium, and low; Weekday is a list which includes Monday, Tuesday, and so on.

Source—Person, document or other origin of a given requirement, useful in determining whom to call for questions or whether the requirement needs to be grouped by persons making the demands.

Priority—Statement of relative importance of the requirement, either to the system (mandatory, critical, optional) or to other requirements (high, medium, low). It is good to track the mandatory or high as an indication of how well the system will meet the greatest needs or for metrics concerning compliance.

Assigned to—Who in the organization is responsible for making sure the requirement is met (person's name or organizational name).

Comments—Reviewer's or writer's comments on a requirement.

Difficulty—An indication of the level of effort needed or how hard it will be to implement the requirement (high, medium, low).

Status—Degree of completeness (completed, partial, not started).

Risk—Confidence measure on the likelihood of meeting (or not meeting) a requirement. Could be high, medium, low or one through ten.

Due By—Date the requirement must be provided.

Method of verification—Qualification type to be used to verify that a requirement has been met: analysis, demonstration, inspection, test, and walkthrough.

Level of Test—Describes the verification lifecycle stage at which the requirement is determined to be met: unit test, component, system or product.

Subsystem Allocation—Name of system or subsystem a requirement is to be assigned to (flight control module, wing assembly, passenger cabin).

Test Number—Identification of a test or other method of verification.

E Appendix: Formats of User and System documents

Once requirements data is gathered, it needs to take shape as an information source. One of the best ways of ensuring that the results of the requirements effort are put to good use is by effectively formatting requirements documents. Two of the main documents used to contain requirements are the User and System Requirements documents.

The User Requirements Document (URD)

Provided below is an outline for a User Requirements Document (URD), based on a widely used standard. The vital thing is to communicate the user's needs to the designers. You can always extend or modify a standard document as long as you explain why you are doing so. If the standard does not provide a section for a necessary piece of information, just insert the information.

Product Perspective	Introduce the system and its context
General Capabilities	What & why these capabilities are needed
General Constraints	What & why the constraints are needed
User Characteristics	Who will use the product and when
Operational Environment	Other systems and their interfaces
Assumptions and Dependencies	Assumptions requirements are based on

Capabilities	The *Scenarios*
Constraints	The qualities demanded by users

System Requirements Document (SRD)

Provided below is an outline for a System Requirements Document (SRD), based on a widely used standard. The SRD is much more involved and detailed. This is the first time the *system* takes shape and is defined. Many of the formats of the URD are present, but greater emphasis is placed here on constraints and how the system is to be verified.

Functional Requirements

System Description	Describe the system and its purpose
System Context diagram	Position of system to the rest of world
Functional Breakdown	Core details of functions to be performed
Operating Modes and States	Different operating options and forms the final system will take on

Non-Functional Requirements

Performance	This is often the largest section
External Interface	One for each external interface
Safety	See Appendix D List of possible constraints

Verification Method

Describes how and when system is tested

Trace Back to User Requirements

Relationship on which to base requirement

F Appendix: Typical requirements problems

Below are some problems commonly found with requirements. Be careful to learn these dangers and avoid them on your project.

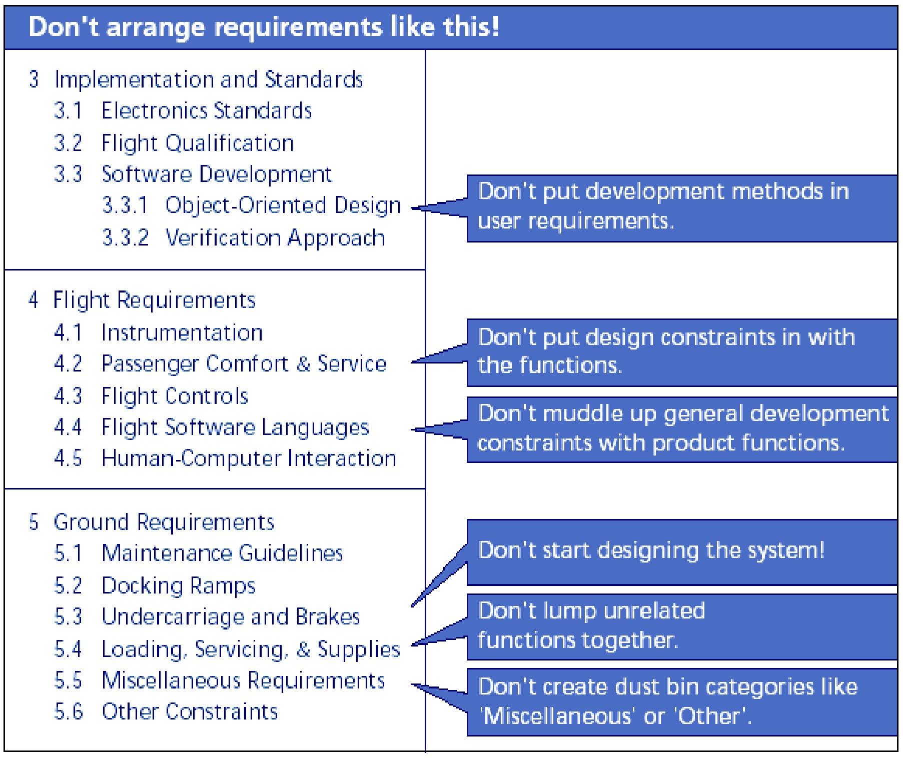
Top 10 requirements problems

Any one of the following 10 problems commonly found could seriously affect your project:

1. Design mixed into the requirements
2. Mixture of user and system requirements
3. Methods and plans in the product requirements
4. No clear owner of the requirement
5. No visible means of testing the requirement
6. Customer too passive or too intrusive
7. Lack of structure: repetition or missing requirements
8. Configuration of requirement version not managed
9. Ambiguity, vagueness, poor wording
10. Commitment to impractical set of requirements

How not to arrange your requirements

The following example shows you how not to arrange your requirements.



Don't arrange requirements like this!

3 Implementation and Standards
- 3.1 Electronics Standards
- 3.2 Flight Qualification
- 3.3 Software Development
  - 3.3.1 Object-Oriented Design
  - 3.3.2 Verification Approach

4 Flight Requirements
- 4.1 Instrumentation
- 4.2 Passenger Comfort & Service
- 4.3 Flight Controls
- 4.4 Flight Software Languages
- 4.5 Human-Computer Interaction

5 Ground Requirements
- 5.1 Maintenance Guidelines
- 5.2 Docking Ramps
- 5.3 Undercarriage and Brakes
- 5.4 Loading, Servicing, & Supplies
- 5.5 Miscellaneous Requirements
- 5.6 Other Constraints

Don't put development methods in user requirements.

Don't put design constraints in with the functions.

Don't muddle up general development constraints with product functions.

Don't start designing the system!

Don't lump unrelated functions together.

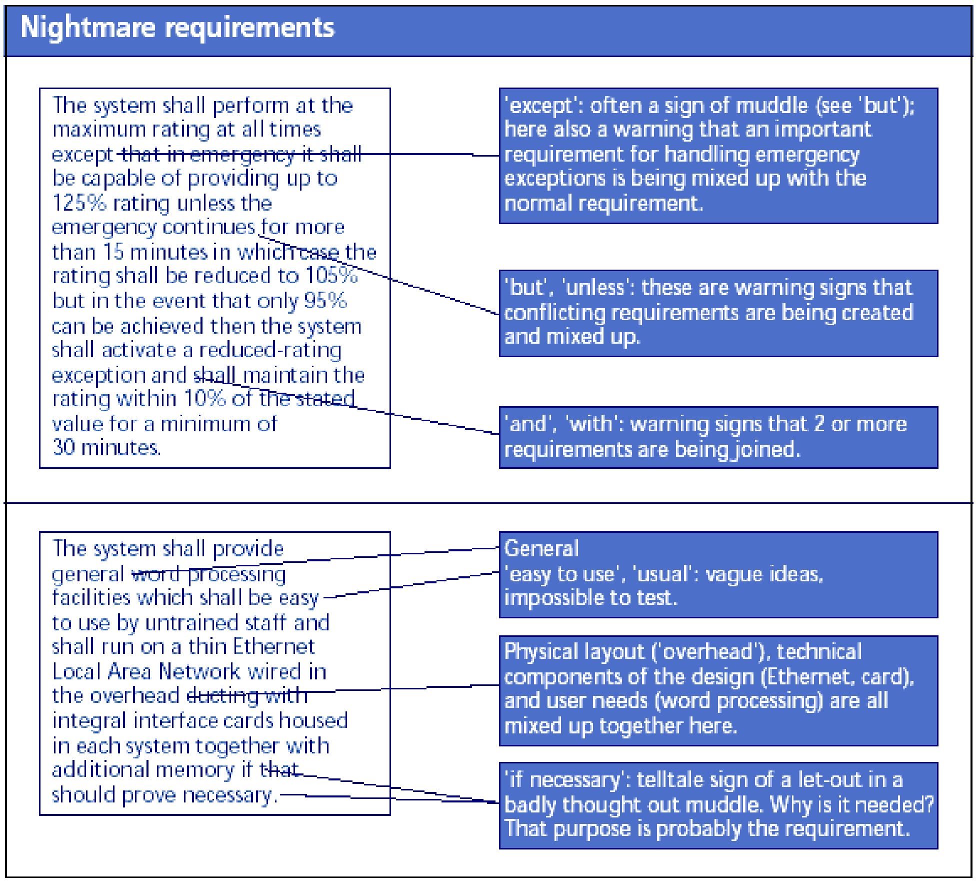
Don't create dust bin categories like 'Miscellaneous' or 'Other'.

In the real world, requirements are sometimes organized very confusingly. If you see a document like the one above, you'll know you have some serious work to do.

Work out which documents each of the headings should be in. Hint: likely candidates are the User Requirements, System Requirements, Architectural Design, Development Plan, and Maintenance Plan (among others).

Nightmare requirements

Many people can unknowingly find themselves writing contorted and ambiguous requirements. Here are some examples of what can go wrong.



Nightmare requirements

The system shall perform at the maximum rating at all times except that in emergency it shall be capable of providing up to 125% rating unless the emergency continues for more than 15 minutes 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 value for a minimum of 30 minutes.

'except': often a sign of muddle (see 'but'); here also a warning that an important requirement for handling emergency exceptions is being mixed up with the normal requirement.

'but', 'unless': these are warning signs that conflicting requirements are being created and mixed up.

'and', 'with': warning signs that 2 or more requirements are being joined.

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 in the overhead ducting with integral interface cards housed in each system together with additional memory if that should prove necessary.

General
'easy to use', 'usual': vague ideas, impossible to test.

Physical layout ('overhead'), technical components of the design (Ethernet, card), and user needs (word processing) are all mixed up together here.

'if necessary': telltale sign of a let-out in a badly thought out muddle. Why is it needed? That purpose is probably the requirement.

Common problems with structure

Listed below are some common problems that affect requirement structure and suggested solutions.

Problem	Solution
All readers need to understand the requirements	Write requirements in everyday language
Long list of requirements is impossible to understand	Make a simple structure of chapters and sections to group the requirements

Problem	**Solution**
Requirements don't show what comes first	Organize the chapters and sections in time order
Some requirements can be applied simultaneously, or in any order—a sequence is an unnecessarily tight ordering	Mark whether sections in the structure are sequences, parallels, or alternatives
The basic sequence of requirements doesn't show what to do if something goes wrong	Add a section for each exception, at the place where it could occur in the normal sequence
Some constraints apply to several requirements	Use a requirements tool to link them together
Users find it hard to get an overview of the whole document	All the above solutions, or • Add a simple diagram • Write an overview • Find out why they find it hard and restructure the document

G Appendix: Types of questions to elicit requirements

Listed below are some types of questions you can use to elicit good requirements from the people you interview. Generally, the questioning will proceed from the general to the specific.

Types of Questions	Definition	Usefulness	Dangers
Open Ended	Prompt for a broad response in a general area	Open up a quiet customer	Avoid with customers who will wander
Is Not	Prompt for broad responses on topics not covered before	Obtains completeness	Could imply distrust
Pregnant	Planned silence after which the customer talks first	Open up a quiet customer	Can intimidate
Vanilla	Comment requesting expansion by the customer	Directs statements to more detail	May encourage trivia
Assertion	Statement inviting agreement and expansion	Requests commitment and amplification	Interruptions can intimidate
Negative	Asking for conditions when previous statements become untrue	Fills in gaps and exceptions	May encourage excess detail
Closed-ended	Questions to produce a limited set of answers	Focuses on specifics	Too many are intimidating
Chin first	Phrasing to imply a “correct” answer	Lead customer towards a conclusion	Customer may feel trapped
Summary	Rephrase of prior statements	Test completeness and confirmation	May encourage invention

H

Appendix: Notices

Trademarks