Description

- A large part of RE is **description**
- In particular, we must describe:-
  - the characteristics of the problem domain
  - the behaviour of the solution system

Description

- Quite often, we describe things by way of **analytic models**, for example :-
  - Data flow diagrams
  - Entity relationship diagrams
  - Decision tables
  - Finite State Machines
  - Class diagrams
  - etc.

  Where appropriate, **use them**.
- However, all these models incorporate **text** and, often, we use text (Natural Language, Plain English) alone . .

  and, **with good reason !**

Textual Description

The bad news :-
- Good documentation requires very good writing
- Producing very good writing is difficult
- At the least, it requires years of practice
- To be blunt, some people will never get there

The good news :-
- But all can improve (!) and
- As we shall see, we have some guidelines (largely from [KOVITZ99])

Textual Description

Jackson identifies 2 ways of defining terms :-
- **Designation**
  - **the informal definition of fundamental terms**
- **(Formal) Definition**
  - **the more or less formal definition of derived terms (in terms of designated terms)**

And (using defined terms) we can produce 2 sorts of description:-
- **Rough sketch**
  - **an informal description (generally, "work in progress")**
- **Refutable description**
  - **a description that is, in principle, testable**

Designation

- The **foundations** for **all other descriptions**
- Takes the form of a **recognition rule**, e.g. -

  call-button := a press button situated in the foyer outside the lift that can be pressed in order to summon a lift
- Most dictionary "definitions" are designations
  - always rely upon a **common understanding**
  - ultimately, tend to be **circular**

  E.g. *

  object – any thing which can be seen, touched or perceived by any of the senses

  thing – any material object (!)

* These and several other definitions are taken from Heineman English Dictionary, 1993

Designation

When and what do we designate?
- **When precision is important (pretty much always in RE !)**
- **We designate all fundamental specialist terms and concepts (that cannot be formally defined)**
- **In particular we must designate data**
  - **inputs**
  - **outputs**
  - **stored data**
- **But other concepts may also require explanation**
  - **states (of the PD and SS)**
  - **problem domain jargon**
  - **non-standard use of terms**

Designation mechanisms

- *equivalence*; means the same as . . . . . e.g.
  - **gloaming** - twilight
  - **parsimonious** - mean
- *classification and discrimination* (*classic*) e.g.
  - **sea-dog** – a sailor with many year's experience.
  - (also, see call-button example above)
- *sum of parts* or *component list* e.g.
  - **boat-details** – the boat's class, its sail number, handicap and owner's name
- *by example*, (usually used to reinforce another (poor) definition) e.g.
  - **going** - the condition of some thing (**eg. the path was rough going**)

Designation

- Designations are inherently informal
- Rely upon some pre-existing common understanding
- Stop when you are confident that there will be too little misunderstanding to cause significant problems.

Jackson also points out there is a big difference between :-
- Defining things that already exist (typically in the PD) and
- Defining things that, as yet, do not (typically in the SS).

You must be even more careful with the latter !

- Designations should be kept to the minimum -
  - wherever possible, use formal definitions instead.

(Formal) Definition

The principles can be applied to NL, but FD benefits from **formal notations.** 3 are suggested :-
- predicate logic (Other Ian's bit !)
- programming languages (as in data declarations)
- (Extended) Backus Naur Form ((E)BNF)

- Relatively few **mechanisms** are needed (or known)
  - some are formalisations of designation mechanisms
  - eg *component parts* :-

    boat ::= boat-name + sail-number + boat-class-name + helm-name;

    (where all the elements on the right have been designated)
  - another is *either or*, for example :-

    race-class ::= boat-class | race-class-name;

(Formal) Definition

More complex, are formal definitions of **relationships** (quick advert for VDM !)

Suppose that if a boat enters a series then it is automatically entered into every race in that series.

We may **designate** a race entry and a series race:-

boat b is entered in series s ≈ **series-entry** (b, s)

race r is part of series s ≈ **series-race** (r, s)

We *could* also designate a race entry thus :-

boat b is entered in race r ≈ **race-entry** (b, r)

But a **formal definition** is better :-

$\forall$ b, r, s • ((series-entry (b, s)) $\wedge$ (series-race (r, s)) $\rightarrow$ (**race-entry** (b, r))

(This can be read as; for any boat, race and series, if the boat is entered in the series and the race is part of the series, the boat will be entered in the race.)

Designation and Definition

Notes:

- neither designations nor formal definitions can be described as **true or false** -

  we are simply stating what we mean by the terms
- they can, however, be **poorly written** and lack clarity and, hence, be open to misunderstanding.
- not all useful information is amenable to formal definition.
  - Eg "the helm is likely to be the owner"

Description

Designations and definitions, are a means to an end; they provide the **building blocks -** to build **useful descriptions**

Things to describe :-
- the **problem domain**
- the **requirements** (the effects that are required to be produced within the PD)
- the **behaviour** of the envisioned **solution system**

This may be generalised to descriptions of :-
- **systems** in terms of their **components** or sub-domains
- the **relationships** between (sub)domains in terms of the **shared phenomena** (often data flows)
- system **behaviour** in terms of functions and, ultimately, relationships between inputs and outputs

Description

There is, as yet, no comprehensive mapping between the various elements needing description and possible description mechanisms

However, there are a few useful constructions :-

- the classic designation
- definition in terms of component parts
- the function statement (later !)

Rough sketches

- "Ordinary English" descriptions
- A useful starting point
- Hopefully, develop into refutable descriptions (but quite often don't)

Refutable descriptions

Once terms have been designated or defined, we can construct **refutable descriptions**. Eg;

- "A **boat-class** always has an associated **PY handicap**."

- How could this be refuted ?
- But we must be careful. Eg;
  - "The entry sheets are then taken back to the club where the race officer enters the details on the results sheets and then works out the results."

Suppose a race officer works out the results and then enters the details on the results sheets? Is that allowed? And what are "the details" anyway? And suppose it was done on the way back to the club?

- Constructing refutable descriptions is very much an **art** !

Writing guidelines (Natural language)

- **Write for the reader** – assume the reader is intelligent, and co-operative.
- **Constantly ask :-**
  - Is there an **easier to understand** alternative expression ?
  - Overloading with **too much information** at once ?
  - What is **most important**, less important to them ?
  - **Too abstract** without illustration ?
  - **Too disconnected** without underlying principle ?
  - **Any reasonable misinterpretations possible ?**
  - **Any benefit** to the reader in this part ?
  - What is the **feel** – formal, stuffy, rambling etc. ?
  - Is the document **boring** ?

Writing guidelines

- Select technology carefully
  - Use lists (like this !)
  - Allow form to follow content (do not force fit)
  - Use pre-defined contents lists only as a guide
- Organise carefully (see also stuff on structure)
  - A place for every detail – every detail in its place (are tidy people the best documenters ?)
  - Provide navigation clues
  - Reinforcement, not repetition
    A trade-off – avoid redundancy in general (why ?) but use overviews, examples etc. to aid understanding

Writing guidelines

- **Avoid Decoy text:**
  - **Metatext** – text that describes the text (eg. "This document is the requirements document for the XYZ lift controller system" (which should be clear from the title)
  - **Generalities** Eg. :- "All requirements should be testable." We already know that's a good idea but is it feasible ? And what if it's not ?
  - **Inclusions** Don't copy other documents – reference them
  - **Duckspeak** Meaningless padding (eg. "The lift request validation function will validate lift requests.")

Writing guidelines - the Naming Rules !

1. **As far as is sensible, adopt the terminology of the problem domain**
2. **Never use the same term for different things**
3. **As far as is sensible, always use the same term for the same thing**
4. **Define all "special" terms (see DD, later)**
5. **Don't invent unnecessary terms (including acronyms!)**

Writing guidelines

- **Examples ????**

Unnatural Language

Structured English – tends to be short, precise sentences, formatted in a similar way to program code

For example :-

2.1 The user is prompted to enter their old PIN
2.2 The user enters their old PIN
2.3 If the old PIN is incorrect :-
2.4 The message "incorrect PIN" is displayed for 4 seconds
2.5 The system returns to the main menu display
2.6 If the old PIN is correct :-
2.7 The user is prompted to enter the new PIN
etc..

A similar style is often found in **use-cases**

Unnatural Language

Pseudo code – even more "codey"

For example :-

2.1 output_message ("enter old PIN")
2.2 input (<user pin>)
2.3 if <user_pin> ≠ <stored_pin> then
2.4 output_message ("incorrect PIN")
2.5 wait (4 seconds)
2.6 goto 2.1
2.7 else
2.8 output_message ("enter the new PIN")
etc.

Unnatural Language

(Extended) Backus Naur Form ((E)BNF)

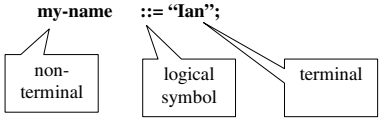
- provides a well-defined grammar for precise and concise definitions
- So far I have concentrated upon **semantics** (meaning)
  BNF helps there, but particularly useful for defining **syntax**
- Syntactic definitions are built upon terminals (aka. literals or primitives)

Unnatural Language

(Extended) Backus Naur Form ((E)BNF)

- A terminal is shown in double quotes, eg. **"A"**
- Non-terminals, are defined using a **defining expression** (or **rule**) composed of :-
  - terminals
  - logical symbols
  - defined non-terminals

Eg; **my-name ::= "Ian";**



EBNF

Symbol	Meaning
= or **:=** or **::=**	is defined as
, or **+**	is followed by (and)
|	exclusive or
{a}	a is repeated (zero or more times)
^{n}{a}	a is repeated at least n times
{a}^{m}	a is repeated up to m times
^{n}{a}^{m}	a is repeated between n and m times (inclusive)
[a]	zero or one occurrences of a (optionality)
. .	indicates a contiguous range (as in "a" . . "z")
( )	delimits a group
(* *) or **/* */**	delimits a comment
;	terminates a definition

EBNF

Example:-

boat := boat-name, boat-class-name, sail-number, helm;
boat-name := {alphanumeric}^{25};
boat-class-name := {alphanumeric}^{10};
sail-number := {digit}^{6};
helm := {alphanumeric}^{25};
alphanumeric := "a" . . "z" | "A" . . "Z" | digit | "-" | "." | " " ;
digit := "0" . . "9" ;

Exercise:- Which of the following are valid values of boat?

- Trapeze, TS-240, 23, Ian K Bray
- D'eau, solo, 3575, Ian K Bray
- Bori Free, spirit, K24570, Fred Basset

EBNF

EBNF comments can be used for semantic definitions (designations)

For example:-

sail-number ::= {digit}^{6};
::= (* the unique identifier of a boat within its boat-class *)

Generally, this is the only kind of "double definition" (syntactic + semantic) that should be used

The Data Dictionary

- **Definitions of all relevant data items (syntax and semantics)**
- **A convenient place to also include designations and definitions of:-**
  - **Problem domain jargon**
  - **Problem sub-domains/sub-systems**
  - **etc.**

The naming rules (version 2!)

1.Designate or, wherever possible, formally define all terms that might give rise to misunderstanding.

2.As far as is sensible, adopt the terminology of the problem domain. (By all means clarify the meaning of existing terms but do not invent new terminology (including acronyms!) just for the sake of it.)

3.Never use the same term for different things. (If this already happens in the PD, you must invent new terms or number them. E.g. in the YRR PD, 'class' is used for both 'boat-class' and 'race-class'.)

4.As far as is sensible, always use the same term for the same thing. (If duplicates already exist, show the equivalence in the data dictionary.)

Validation

- Checks that the right product is being built
- Helps ensure delivery of what the client wants
- highly pertinent to RE

Need to be performed at every stage during the process:

Elicitation

checking back with the elicitation sources "So, are you saying that . . . . . ?"

Analysis

checking that the PD description and requirements are correct

Specification

checking that the defined behaviour will meet the requirements

(And, after RE, checking that the design will produce the defined behaviour and, ultimately that the system does behave as required.)

Techniques:

- Simple checks
- Review
- Logical analysis
- Prototypes and enactments
- Functional test design
- User manual development

Simple checks

Eg.:-

- On completion of the RD check through the elicitation notes and ensure that nothing is omitted
- On completion of the specification check that all requirements have been reflected

Review

- Review of RE deliverables
- Various types of review – an example later!

Logical analysis

Requires a **formal specification** (VDM, FSM . . . .)

Eg. FSM, can check:-

- that all states are reachable and leavable
- every transition has exactly one trigger
- that every trigger (event) is recognised
- etc.

Prototypes and enactments

- Very good for checking with users, client etc. (Easier to relate to than specifications.)
- Can use paper-based or software-based prototypes
- Walk through various scenarios (can use use-cases)

Functional test design

- System level functional tests must be designed sooner or later.
- Can (should) be derived from the specification
- Designing the tests can reveal errors in the specification (prior to system design and build!)

User manual development

- Very similar considerations to FTD (above)

Requirements Validation

Validation objectives

- Certifies that the requirements document is an acceptable description of the system to be implemented
- Checks a requirements document for
  - Completeness and consistency
  - Conformance to standards
  - Requirements conflicts
  - Technical errors
  - Ambiguous requirements

Analysis and validation

- Analysis works with raw requirements as elicited from the system stakeholders
  - "Have we got the right requirements" is the key question to be answered at this stage
- Validation works with a final draft of the requirements document i.e. with negotiated and agreed requirements
  - "Have we got the requirements right" is the key question to be answered at this stage

Validation inputs and outputs

Validation inputs

- Requirements document
  - Should be a complete version of the document, not an unfinished draft. Formatted and organised according to organisational standards
- Organisational knowledge
  - Knowledge, often implicit, of the organisation which may be used to judge the realism of the requirements
- Organisational standards
  - Local standards e.g. for the organisation of the requirements document

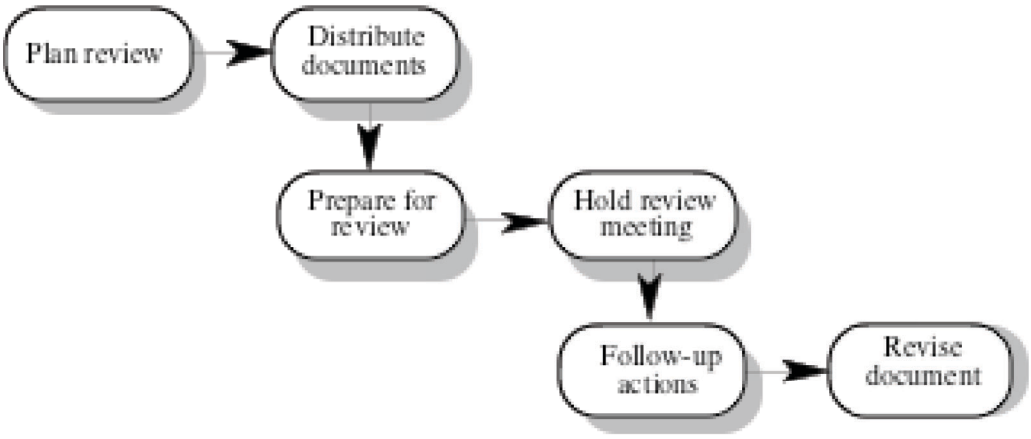
Validation outputs

- Problem list
  - List of discovered problems in the requirements document
- Agreed actions
  - List of agreed actions in response to requirements problems. Some problems may have several corrective actions; some problems may have no associated actions

Requirements reviews

- A group of people read and analyse the requirements, look for problems, meet and discuss the problems and agree on actions to address these problems

Requirements review process



Review activities

- Plan review
  - The review team is selected and a time and place for the review meeting is chosen.
- Distribute documents
  - The requirements document is distributed to the review team members
- Prepare for review
  - Individual reviewers read the requirements to find conflicts, omissions, inconsistencies, deviations from standards and other problems.

Review activities

- Hold review meeting
  - Individual comments and problems are discussed and a set of actions to address the problems is agreed.
- Follow-up actions
  - The chair of the review checks that the agreed actions have been carried out.
- Revise document
  - The requirements document is revised to reflect the agreed actions. At this stage, it may be accepted or it may be re-reviewed

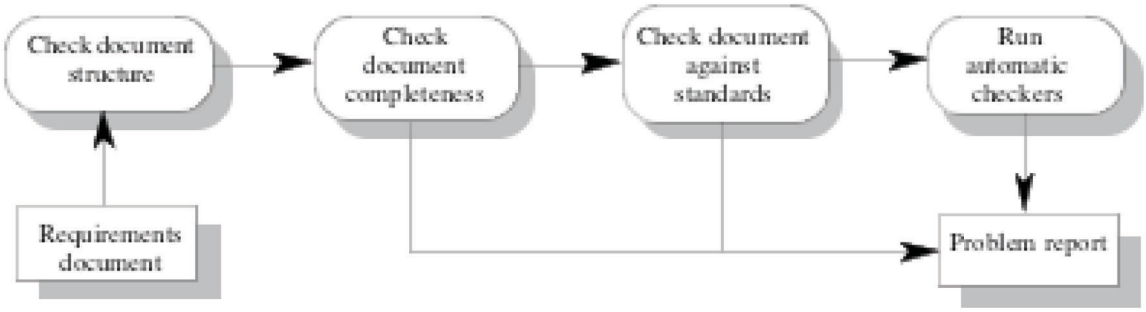
Problem actions

- Requirements clarification
  - The requirement may be badly expressed or may have accidentally omitted information which has been collected during requirements elicitation.
- Missing information
  - Some information is missing from the requirements document. It is the responsibility of the requirements engineers who are revising the document to discover this information from system stakeholders.
- Requirements conflict
  - There is a significant conflict between requirements. The stakeholders involved must negotiate to resolve the conflict.
- Unrealistic requirement
  - The requirement does not appear to be implementable with the technology available or given other constraints on the system. Stakeholders must be consulted to decide how to make the requirement more realistic.

Pre-review checking

- Reviews are expensive because they involve a number of people spending time reading and checking the requirements document
- This expense can be reduced by using pre-review checking where one person checks the document and looks for straightforward problems such as missing requirements, lack of conformance to standards, typographical errors, etc.
- Document may be returned for correction or the list of problems distributed to other reviewers

Pre-review checking



Review team membership

- Reviews should involve a number of stakeholders drawn from different backgrounds
  - People from different backgrounds bring different skills and knowledge to the review
  - Stakeholders feel involved in the RE process and develop an understanding of the needs of other stakeholders
- Review team should always involve at least a domain expert and an end-user

Review checklists

- Understandability
  - Can readers of the document understand what the requirements mean?
- Redundancy
  - Is information unnecessarily repeated in the requirements document?
- Completeness
  - Does the checker know of any missing requirements or is there any information missing from individual requirement descriptions?
- Ambiguity
  - Are the requirements expressed using terms which are clearly defined? Could readers from different backgrounds make different interpretations of the requirements?

Review checklists

- Consistency
  - Do the descriptions of different requirements include contradictions? Are there contradictions between individual requirements and overall system requirements?
- Organisation
  - Is the document structured in a sensible way? Are the descriptions of requirements organised so that related requirements are grouped?
- Conformance to standards
  - Does the requirements document and individual requirements conform to defined standards? Are departures from the standards, justified?
- Traceability
  - Are requirements unambiguously identified, include links to related requirements and to the reasons why these requirements have been included?

Checklist questions

- Is each requirement uniquely identified?
- Are specialised terms defined in the glossary
- Does a requirement stand on its own or do you have to examine other requirements to understand what it means?
- Do individual requirements use the terms consistently
- Is the same service requested in different requirements? Are there any contradictions in these requests?
- If a requirement makes reference to some other facilities, are these described elsewhere in the document?
- Are related requirements grouped together? If not, do they refer to each other?

Requirements problem example

- "4. EDDIS will be configurable so that it will comply with the requirements of all UK and (where relevant) international copyright legislation. Minimally, this means that EDDIS must provide a form for the user to sign the Copyright Declaration statement. It also means that EDDIS must keep track of Copyright Declaration statements which have been signed/not-signed. Under no circumstances must an order be sent to the supplier if the copyright statement has not been signed."

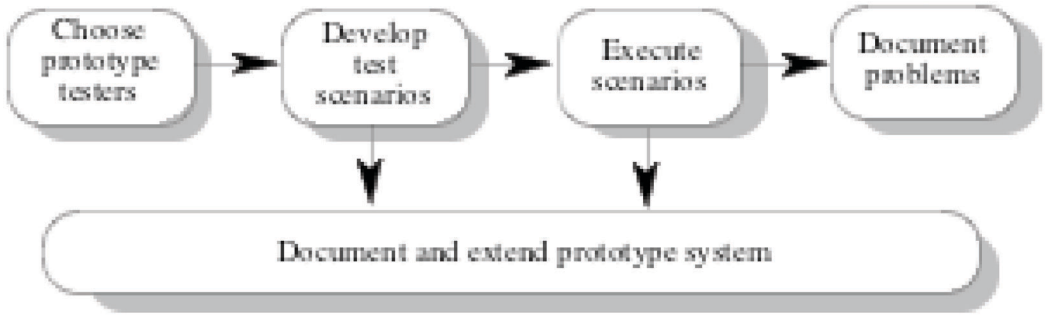
Problems

- Incompleteness
  - What international copyright legislation is relevant?
  - What happens if the copyright declaration is not signed?
  - If a signature is a digital signature, how is it assigned?
- Ambiguity
  - What does signing an electronic form mean? Is this a physical signature or a digital signature?
- Standards
  - More than 1 requirement. Maintenance of copyright is one requirement; issue of documents is another

Prototyping

- Prototypes for requirements validation demonstrate the requirements and help stakeholders discover problems
- Validation prototypes should be complete, reasonably efficient and robust. It should be possible to use them in the same way as the required system
- User documentation and training should be provided

Prototyping for validation



Prototyping activities

- Choose prototype testers
  - The best testers are users who are fairly experienced and who are open-minded about the use of new systems. End-users who do different jobs should be involved so that different areas of system functionality will be covered.
- Develop test scenarios
  - Careful planning is required to draw up a set of test scenarios which provide broad coverage of the requirements. End-users shouldn't just play around with the system as this may never exercise critical system features.
- Execute scenarios
  - The users of the system work, usually on their own, to try the system by executing the planned scenarios.
- Document problems
  - Its usually best to define some kind of electronic or paper problem report form which users fill in when they encounter a problem.

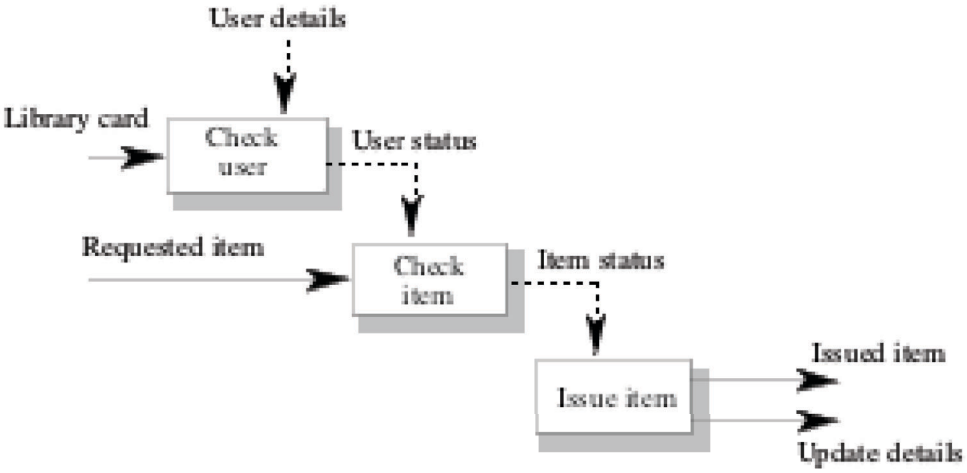
User manual development

- Writing a user manual from the requirements forces a detailed requirements analysis and thus can reveal problems with the document
- Information in the user manual
  - Description of the functionality and how it is implemented
  - Which parts of the system have not been implemented
  - How to get out of trouble
  - How to install and get started with the system

Model validation

- Validation of system models is an essential part of the validation process
- Objectives of model validation
  - To demonstrate that each model is self-consistent
  - If there are several models of the system, to demonstrate that these are internally and externally consistent
  - To demonstrate that the models accurately reflect the real requirements of system stakeholders
- Some checking is possible with automated tools
- Paraphrasing the model is an effective checking technique

Data-flow diagram for Issue



Paraphrased description

Check user	
Inputs and sources	User's library card from end-user
Transformation function	Checks that the user is a valid library user
Transformation outputs	The user's status
Control information	User details from the database
Check item	
Inputs and sources	The user's status from Check user
Transformation function	Checks if an item is available for issue
Transformation outputs	The item's status
Control information	The availability of the item
Issue item	
Inputs and sources	*None*
Transformation function	Issues an item to the library user. Items are stamped with a return date.
Transformation outputs	The item issued to the end user Database update details
Control information	Item status - items only issued if available

Requirements testing

- Each requirement should be testable i.e. it should be possible to define tests to check whether or not that requirement has been met.
- Inventing requirements tests is an effective validation technique as missing or ambiguous information in the requirements description may make it difficult to formulate tests
- Each functional requirement should have an associated test

Test case definition

- What usage scenario might be used to check the requirement?
- Does the requirement, on its own, include enough information to allow a test to be defined?
- Is it possible to test the requirement using a single test or are multiple test cases required?
- Could the requirement be re-stated to make the test cases more obvious?

Test record form

- The requirement's identifier
  - There should be at least one for each requirement.
- Related requirements
  - These should be referenced as the test may also be relevant to these requirements.
- Test description
  - A brief description of the test and why this is an objective requirements test. This should include system inputs and corresponding outputs.
- Requirements problems
  - A description of problems which made test definition difficult or impossible.
- Comments and recommendations
  - These are advice on how to solve requirements problems which have been discovered.

Requirements test form

Requirements tested: 10.(iv)

Related requirements: 10.(i), 10.(ii), 10.(iii), 10.(vi), 10.(vii)

Test applied: For each class of user, prepare a login script and identify the services expected for that class of user.

The results of the login should be a web page with a menu of available services.

Requirements problems: We don't know the different classes of EDDIS user and the services which are available to each user class. Apart from the administrator, are all other EDDIS users in the same class?

Recommendations: Explicitly list all user classes and the services which they can access.

Hard-to-test requirements

- System requirements
  - Requirements which apply to the system as a whole. In general, these are the most difficult requirements to validate irrespective of the method used as they may be influenced by any of the functional requirements. Tests, which are not executed, cannot test for non-functional system-wide characteristics such as usability.
- Exclusive requirements
  - These are requirements which exclude specific behaviour. For example, a requirement may state that system failures must never corrupt the system database. It is not possible to test such a requirement exhaustively.
- Some non-functional requirements
  - Some non-functional requirements, such as reliability requirements, can only be tested with a large test set. Designing this test set does not help with requirements validation.

Key points

- Requirements validation should focus on checking the final draft of the requirements document for conflicts, omissions and deviations from standards.
- Inputs to the validation process are the requirements document, organisational standards and implicit organisational knowledge. The outputs are a list of requirements problems and agreed actions to address these problems.
- Reviews involve a group of people making a detailed analysis of the requirements.
- Review costs can be reduced by checking the requirements before the review for deviations from organisational standards. These may result from more serious requirements problems.

Key points

- Checklists of what to look for may be used to drive a requirements review process.
- Prototyping is effective for requirements validation if a prototype has been developed during the requirements elicitation stage.
- Systems models may be validated by paraphrasing them. This means that they are systematically translated into a natural language description.
- Designing tests for requirements can reveal problems with the requirements. If the requirement is unclear, it may be impossible to define a test for it.