
# What do you mean?

R.D. McDowall

Own up, we've all done it. It happens in a number of ways; you could be looking through a computer magazine and see an advert for the latest software package. Before you know it you're on autopilot: pick up the phone with your credit card in hand and the next day you're installing the package on your PC. Does it work as you expect? Probably not! Take comfort in the fact that you are not alone. You are not the first and you definitely won't be the last.

What about the waste of software at a higher level? What about your organization? Every company has a horror story that it keeps quiet about, but the staff know how much money and time were wasted on a software or computer project. Rumour has it that some companies have more than one data-management project that has gone down the tubes.

What about the waste of scientific data systems on an even larger scale? Governments certainly have larger budgets and can make larger mistakes. A study of US government software contracts in the 1980s was very enlightening[1]. This looked at nine projects with a total value of US$6.75 million. The analysis showed that of the taxpayers' money spent, the software that was:

Never delivered = $2 900 000
Never used = $3 200 000
Unaccounted = $350 000
Used after modifications = $200 000
Used as delivered = $100 000

Thus, less than 2% of the total value of all of the software was used as delivered. (And you thought your company was bad at delivering software projects.) Why was so much money wasted? The US study gives us a clue. When the reasons for failure were examined, three causes emerged:

- poor or misunderstood user requirements
- inability to cope with new requirements (inflexible)
- poor or no user documentation.

Of the three, the majority (65%) was the result of poor or misunderstood user requirements.

Therefore, this issue's V&V column will examine the role of the user requirements specification (URS) in defining what the end user needs and wants from a software package. Before we look in detail at the URS, we need to stand back and have an overview of the whole system develop-ment life cycle to ascertain where this document fits in and how it drives and controls the whole process. In future V&V columns, we shall also return to discuss various aspects of the life cycle.

# It doesn't work?

## Systems Development Life Cycle (SDLC)

The development of any computer system or software package should follow a life cycle. There are different models available, such as a waterfall. Figure 1 is based upon the ISO V model. The activities on the left represent the design activities and those on the right the testing activities to ensure that the design has been achieved.

The different stages of the life cycle are:

- Concept: the idea for the system.
- Requirements: what the users of the system want from the delivered system.
- Design: designing the software modules and overall system architecture.
- Build: programming the modules.
- Test: testing the modules; assembling and testing the modules, units and their integration; and structural testing the whole system.
- Qualify or User-Acceptance tests: check that the delivered system matches the user requirements.
- Maintain: operate, enhance and maintain the system (this may repeat some or all of the phases of the life cycle).
- Retire: Planned retirement of the system and transition or archive of the data and users to the replacement system.

## Straight flow or feedback loops?

What is not shown on the diagram are the feedback loops between the different life-cycle phases. There is not usually a clean cut-off between each phase of the life cycle (i.e., one phase ends and the next one starts). Usually, there are questions raised in the next or later phases, if the information or description written down in an earlier phase is not clear enough. We shall return to this later in this column.

Obviously if there are a sufficient number of problems it will delay the project and increase the cost. This is compounded by the fact that the further a problem or unresolved issue goes through the life cycle, the greater its impact and the cost of resolution. An alternative scenario, especially when users are not available, is that the programmer interprets what he or she thinks the user wants, to save time. Inevitably, this is wrong and leads to further problems.

Therefore there are two issues:

- The need to specify the system requirements in sufficient detail to resolve ambiguities
- Writing the requirements in a language that can be understood by both users and computing professionals.

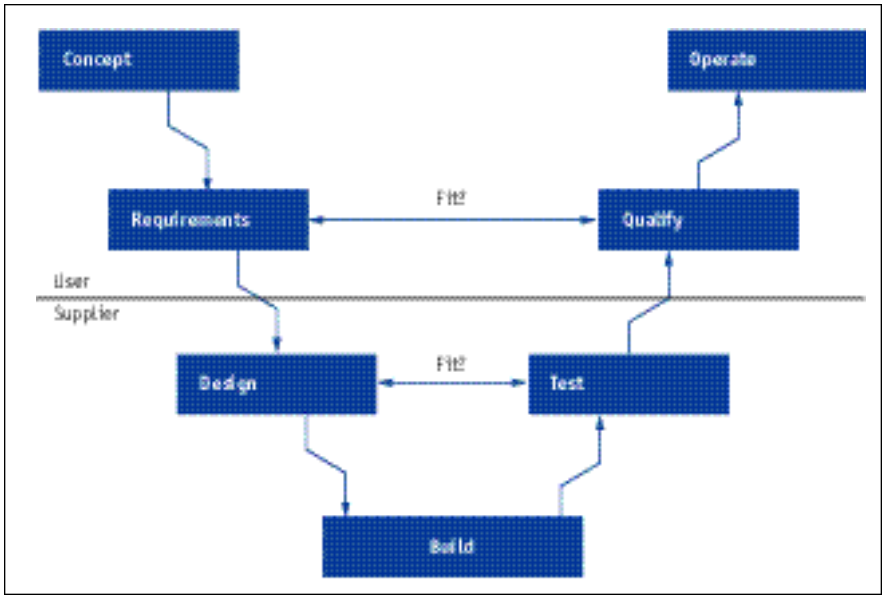


figure 1 The System Development Life Cycle.

## The way forward

The way of meeting both requirements is via a user requirements specification (URS). This document is your map and guide through the SDLC. Without it you are lost. Without sufficient detail, you take a very slow and very expensive scenic tour through the life cycle. For your sanity and your company's cash flow, the better the URS, the quicker the system will be to develop, or select, and then implement. The best advice, therefore, is to spend as much time on the URS as possible; it will ensure the best payback.

Traditionally, the spend distribution on a project is about 10% of the cost on designing, 40% on developing and getting the system running, and 50% on supporting it throughout the operating life. More time spent on the design will reduce the time and resources spent on the operation phase making it more cost-effective. This, in turn, will reduce the overall operating costs of the system.

A well-written URS provides several specific benefits[2-3]. This is because it:

- Serves as a reference against which off-the-shelf commercial products are selected, evaluated in detail, and any enhancements defined.
- Reduces the system-development effort and costs, as careful review should reveal omissions, misunderstandings and/or inconsistencies in the early development or selection phases, at a time when these errors are easier and less expensive to correct.
- Provides the input to user acceptance test specifications and/or qualification of the system.
- Provides input to the implementation plan as the project team will know the type of system, either commercial or do-it-yourself, that will be chosen.
- Facilitates a controlled and verifiable continual process of enhancement.

**URS should represent a binding agreement between the customer and the developer or vendor about the overall characteristics of a computerized system.**

## General guidelines for a URS

A URS defines clearly and precisely what the customer wants the system to do, and should be understood by both the customer and the supplier. In this context the customer can be a company or the user, and the supplier can be an internal programming or computer group, or a commercial supplier. For some systems, the customer and supplier can be the same organizational unit or individual. The URS is a 'living' document, and must be kept updated according to a change control procedure throughout the computer system life cycle.

A URS defines the functions to be performed, the data on which the system will operate, and the operating environment. The document also defines any non-functional requirements, constraints such as time and costs, and what deliverables are to be supplied. The emphasis is on the required functions and not the method of implementing those functions, as this may be the identification of a solution.

If helpful, the requirements entered in a URS may be based on experience of a prototype system. Some characteristics of the final system can then be taken directly from the prototype, whereas other requirements can be ascertained by running experiments on the prototype. Prior to authorization, the URS will often undergo refinement. Selection of capable suppliers and vendors is critical at this stage.

When associated with an Invitation to Tender, the URS should be free from proprietary information or technology-driven requirements. The URS must be reviewed to meet standards when reaching completion.

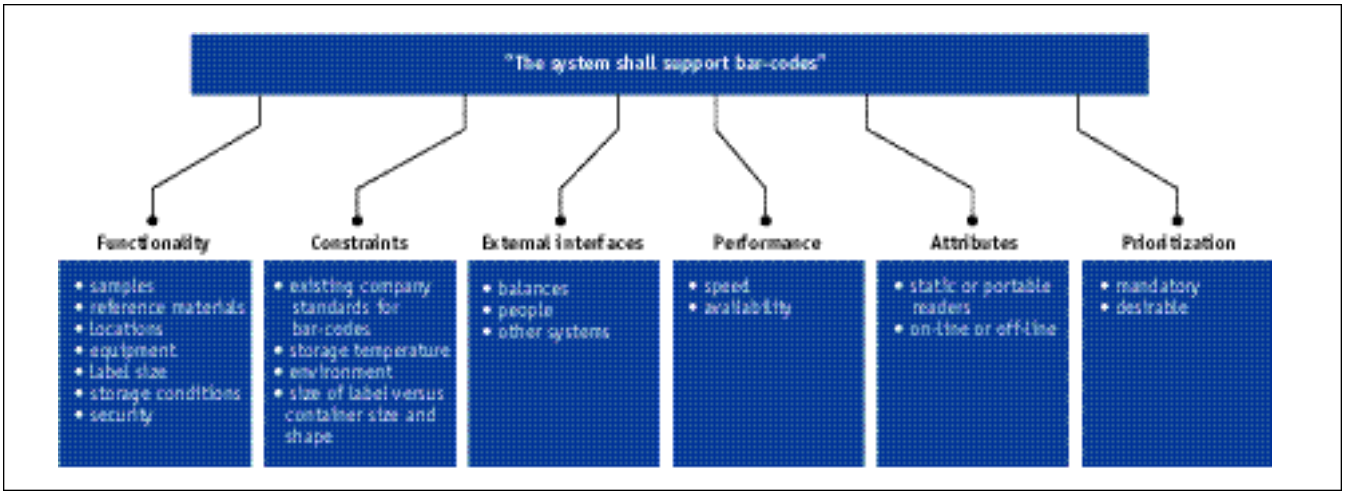


figure 2 Breaking a concept into a user requirement.

## Nature of the URS

The following basic issues should be addressed:

- Functionality: What is the system or function supposed to do?
- External interfaces: How does the system interact with users, hardware or software?
- Performance: What is speed, availability, response time, etc. of the various functions of the system?
- Attributes: What considerations are given to portability, correctness, maintainability, security, etc?
- Design constraints imposed on an implementation: Are there any required standards in effect, resource limits, etc?
- Prioritization: All requirements suggested by various users should be ranked for importance. Are the requirements essential (i.e., are they compulsory for the system to operate?) or desirable (i.e., simply nice to have?). Will they be permanent features throughout the lifetime of the system, or dependent on future events or scientific equipment?

The URS should represent a binding agreement between the customer and the developer or vendor about the overall characteristics of a computerized system. However, this rarely happens and the purchasers can leave themselves open to poor delivery times or a poor quality product.

## Writing the specification

These guidelines should be followed during the production of the specification[2-3]:

- Each requirement statement should be uniquely referenced and no longer than 250 words.
- The URS should be consistent. Therefore, requirement statements should not be duplicated or contradicted.
- Express requirements and not design solutions.
- Each requirement should be testable (this allows the tests to be written as soon as the URS is finalized).
- The document must be understood by both customer and supplier. Thus, ambiguity and jargon should be avoided, or if used, key words should be defined in a specific section in the document.
- Ideally, the requirements should be prioritized as mandatory or desirable.
- The URS should be modifiable. There may need to be a formal review of the URS between the customer and supplier to check understanding and that requirements have been met (or not) in the Functional Specification or Design documents. Changes should be under a formal control procedure to avoid 'creeping functionality' and an unworkable design.
- Any requirement must be traceable to earlier documents and to documents that are derived from the URS (e.g., design documents or testing plans, etc.).

A URS is correct if every requirement stated has only one interpretation and is met by the system. Unfortunately, these are very rare documents.

## Organizing requirements

As the finalized URS tends to be extensive, careful consideration should be given to organizing requirements in the easiest manner to understand. There is no one optimal organization for all types of data-management systems. Different classes of systems lend themselves to different organization of requirements. Some of these are described below[2]:

- System Mode: organize requirements according to system mode (training, service, production, emergency, etc.).
- Workflow: describing the features in relation to the process that you are automating.
- User Classes: organize requirements according to the privileges each user class is assigned (maintenance, operators, etc.).
- Objects: organize requirements in accordance with object attributes (printers, fermentation vessels, etc.).
- Feature: organize requirements to describe the desired services provided by the system (entry of samples to the system, or results, etc.). This is one of the most common ways to organize a URS.
- Stimulus: organize requirements to describe how each stimulus (input) is supposed to be dealt with (power loss, hardware interrupts, software alarms, etc.).
- Response: organize requirements to describe all the functions needed to generate a specific response (output), (generation of a packing list, opening a pressure relief valve, etc.).
- Functional hierarchy: the overall functionality can be arranged into a hierarchy of functions organized by common inputs, common outputs, or common internal data access. Data flow diagrams and data dictionaries can be used to show the relationships within the functions and data.

More than one of these techniques may be used together to clarify what is required, for example, a workflow with security can explain how best to ensure data security and integrity.

## Go with the workflow

The best framework for writing a URS for many data-management systems is to follow the process or workflow that the data system will be automating. Here, there may be some difficulty as most organizational units are based around specific functions, and a process will go across functions. These organizational silos are like medieval empires: this is the job that we do and nothing more, when automating across these boundaries it is inevitable that resistance will be encountered. (This brings one more problem for management and the project team to handle.) Therefore, if you have mapped the process, this makes an ideal prompt for the URS.

Alternatively, if the process is

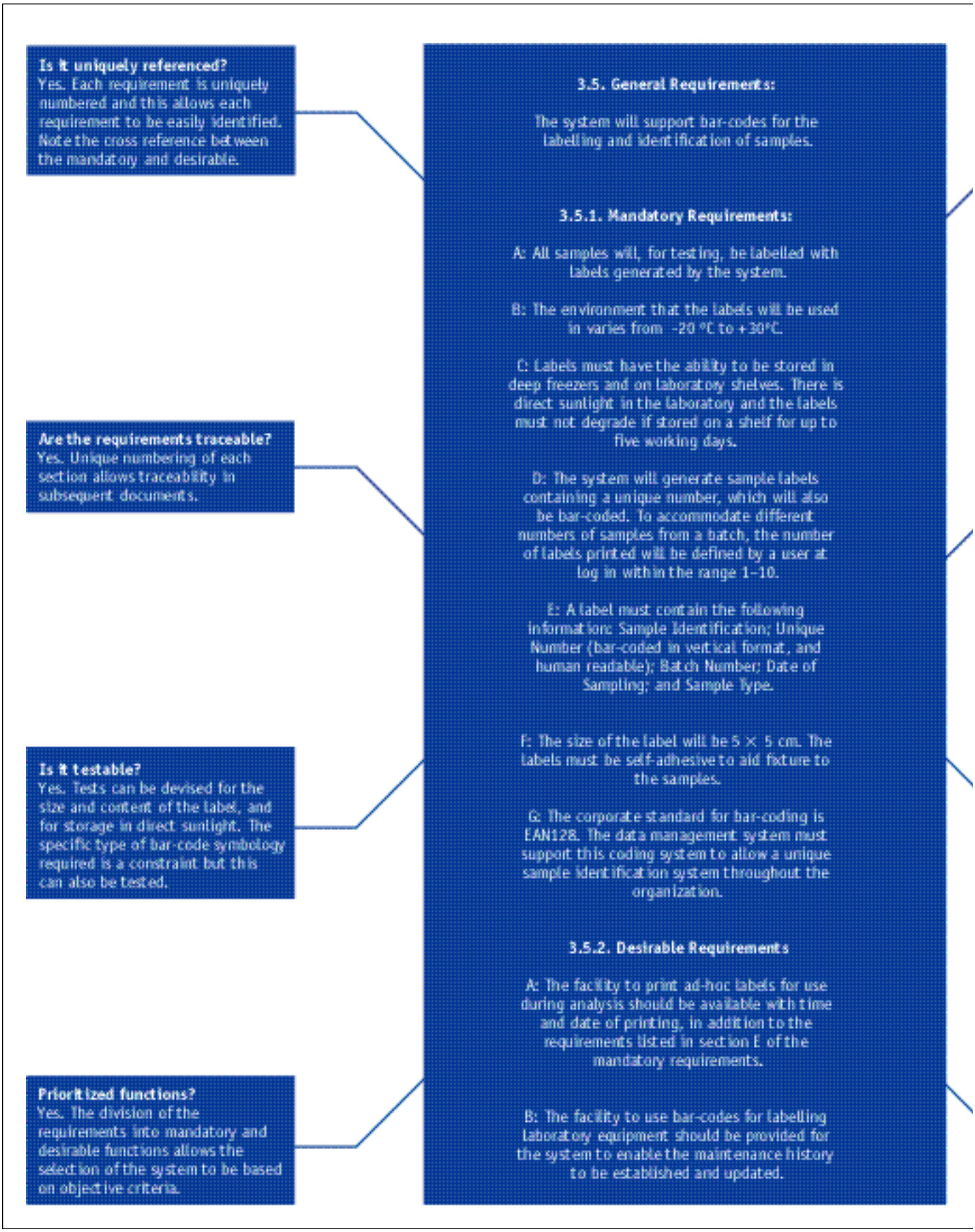


figure 3 Improved user requirements specification.

**Is it modifiable?**
Yes, although each version of the URS should be carefully controlled and authorized, as functional creep is a major problem with data management systems. There must be a formal change control system to prevent any update to the URS being included: each requirements' update must be justified and assessed.

**Does it express requirements?**
Yes. There is no mention of the way that the requirements can be implemented.

**Is it unambiguous?**
Possibly. This is one of the most difficult areas to get right. What appears to be acceptable at first sight may become ambiguous as the data system project progresses. Look carefully at the requirement in 3.5.2.8 for maintenance records of equipment. This sounds fine. As it is a desirable requirement, it may not receive as much detailed discussion as a mandatory requirement. However, where is the detail? What are the maintenance records? How many items of equipment will be required to be entered? Compare this with the requirements for the labels outlined in 3.5.1.E.

**Is it consistent and cannot be contradicted?**
At first sight, the statements appear consistent but see the comments on the maintenance records.

mapped, you have a choice between automating the status quo, or redesigning or simplifying the process before automating. Simplification of the process will allow greater benefits of automation through the use of information technology. Alternatively, if the process is a mess and you automate it, the outcome is very simple: an automated mess. Think carefully before automating the status quo.

- If you are working in a regulated environment, how can you qualify or validate this system?

Let's judge the adequacy of this statement by comparing it with the guidelines listed above:

1. Is it uniquely referenced? No.
2. Is it consistent and cannot be contradicted? No.
3. Does it express requirements? No.
4. Is it testable? No.
5. Is it unambiguous? No.
6. Prioritized function? No.
7. Is it modifiable? Yes!
8. Is the requirement traceable? No.

The key factors to look at are, 'is there sufficient detail?', 'is it unambiguous?', and, 'can it be tested?' Here the requirement fails miserably.

What we have above is a statement of need not a user requirement. This need is capable of many interpretations and also many misinterpretations. It is certainly capable of many misinterpretations that will be the cause of many costly enhancements if implemented in this system.

This general approach has resulted in many poorly designed data-management systems. Put yourself in the position of the programmer or systems analyst responsible for the interpretation of this specification. How far could you progress on writing or programming the system? On the one hand, not very far because there is no detail. Or, alternatively, and here is a major problem, this statement is interpreted by the analyst without reference to the user and you get a totally unsuitable system.

**The use of prototyping or rapid application development (RAD) does not eliminate the need for defining the user requirements. In fact, the use of these methodologies brings users and developers closer together**

## What does this mean in practice?

This idea of documenting what we want in sufficient detail sounds great, but it means more work, doesn't it? Yes, this is true, but consider the benefits. The more time you spend in the specification and design phase getting your ideas and concepts right the quicker the rest of the life cycle will go as you know what you want. You will get a system package that meets your requirements more fully and there will be less discussion later in the life cycle.

Contrast this to a package selection with no user requirements. (This bit should be easy as we have all done it.)

## A specific example

To illustrate the problem, let's look at a real example from a URS. The URS states succinctly:

'The system shall support bar-codes'

This sounds very impressive doesn't it? We can have a scientific system that is up to the technological level of the local supermarket. But wait a minute:

- How do we select a system based on this requirement?
- How do we test the final system when delivered or written?

## How can we get it right?

Using Figure 2 as a reference, let's look and see how we can improve the requirements. In the guide for writing a URS we discussed the following areas that should be addressed to improve the requirement:

1. Functionality: The bar-codes are used by the system to generate labels for samples rather than just 'support bar-codes'. Moreover, are the labels to be limited to just samples? Do you want to include labels for reference materials, locations (both storage locations and the individual shelves inside), or equipment that may need calibration or maintenance as well as samples? It may be that chain of custody is important and location is a requirement that will feature highly in this data system and be linked with time delays or cumulative time. The size of the label should be stated along with the types of information that should be on the label itself. Remember the number of humans that can read a bar-code accurately is relatively low and you should include at least some written information on the label. You must specify this in the URS as this will affect the label stock and the printer type that will be supplied. However, a bar-code is more than a method for sample identification. You can encode data-entry information into a bar-code and use this instead of a keyboard for operating an application. Is this a requirement in this application?
2. Attributes: Do you want static or portable bar-code readers? This will depend on the type of work you are doing. If you need portable bar-code readers, should they be on-line or off-line?

What type of containers will you be sticking the labels on? Is there a flat surface or a curved one?

How robust do the readers need to be? Will they be used in the middle of the North Sea or in a benign laboratory environment? Is contact with the sample acceptable or, in the case of biological samples (such as with hepatitis), to be discouraged?

3. Constraints: There may be a number of constraints on the use of bar-codes. This may stem from the use of existing corporate bar-code standards and will have to be incorporated into the new system. This is a constraint that must be stated. Otherwise, a system could be purchased that does not conform to the corporate standard.

The temperature range that the labels will be used in must be stated as this may be a constraint. Within a relatively narrow temperature of 4–25 ºC there may not be a problem with what is supplied by a vendor. However, if the sample has to be stored frozen at -20, -40, -80 or even -180 ºC, how would this impact the delivered solution?

The environment that the labels will be used in may also be a constraint. For example, condensation, atmospheric conditions, heat and sunlight will affect thermal labels.

4. External interfaces: Are there any other systems that the bar-code labels will need to be interfaced to?
5. Performance: If the bar-code reader is to be on-line the performance of the system will need to be rapid. If the readers will be used off-line, the system performance will not be critical. However, the reader will need to be robust to ensure that data collected off-line are not lost because of reader failure.
6. Prioritization: Is this a requirement that is mandatory or just nice to have?

Having read this section, is the statement: 'The system shall support bar-codes' specific and unambiguous? Of course not! Now you know this, what are you going to do about it?

## A specific example: take two

Figure 3 shows a better URS concerning the use of bar-codes within the data management system. In this example, we judge the adequacy of this updated URS by comparing it with the guidelines listed above.

The URS in Figure 3 is definitely an improvement on the one originally discussed. However, there can still be problems when it comes to interpretation. The above section of the URS is not perfect. We still have to make assumptions about:

- the nature of the surface of the container we are sticking the labels on;
- do we want static or portable readers?
- do we require on-line or off-line readers?
- robustness of the readers could be inferred from the laboratory conditions but should be stated;
- if there is a company-wide system for bar-codes will there be other external systems to interface with?
- no statements of performance have been made.

So what looks to be an improved URS is not ideal but is better than the first attempt.

Ideally, to improve the quality of the draft URS, it should be read and challenged by users and others who have not been involved in the writing of the document. This will take time but, in my opinion, will be time well spent. However, there are alternative ways to ensure the user requirements have been defined. This involves prototyping.

## Impact of prototyping on the URS

The use of prototyping or rapid application development (RAD) does not eliminate the need for defining the user requirements. In fact, the use of these methodologies brings users and developers closer together to define and refine system requirements, which should help reduce the number of systems that are not fit for purpose. Any steps in this direction are to be welcomed and encouraged.

You should realize that when using prototyping, the traditional life-cycle phases are merged and will even overlap.

In some industries, there is a need for a formal proof that the system works as specified, such as in the pharmaceutical and agrochemical industries. Here prototyping is a means of defining requirements, but after the prototype has been finished the specification must be written and approved.

There are two basic approaches to prototyping. The first is to use an unstructured approach to develop user requirements in which functions are developed rapidly. This produces a system that has outline functions from which the URS can be written in the confidence that the requirements match the user expectations. It is important to realize that the programs are not robust and the code will contain many errors. Management must also realize that the system is not complete. The prototype system must be discarded and a new one written based on structured programming standards and formal testing.

The second approach uses structured programming standards from the start and works in an iterative way to develop the final system. There is an initial URS followed by a prototype, this provides the input to update the specification and the development of a new prototype. This goes on until the system is developed. As the system has been written in a structured way, the system is now ready for testing once the URS has been updated to reflect the accepted prototype.

## Role of the URS in validation

In some industries, such as the pharmaceutical, medical device and agrochemical, and in laboratories working under ISO guide 25, software must be validated to demonstrate that it is fit for its intended purpose. One definition of validation is from the Food and Drug Administration (FDA) and this states: "Establishing documented evidence which provides a high degree of assurance that a specific process will consistently produce a product meeting its predetermined specification and quality attributes"[4].

The key concepts in this definition are:

- documented evidence
- high degree of assurance
- consistency and reproducibility
- predetermined specification.

Note the last requirement: predetermined specification. This means that if you work in a regulated industry and don't want to write a URS to get the right system or protect your investment, you'll have to write one to validate the system.

Last year, the FDA published a draft guidance on Principles of Software Validation[5]. This is intended for medical devices but the document describes common principles of software validation. It should also be remembered that the document is intended for medical devices that can have life-threatening consequences with, what a large software company call, a 'feature', and the rest of us call a 'bug' or an 'error'.

The document states succinctly: "To validate software, there must be predetermined and documented user requirements specifications". This is a very clear statement that a URS is a mandatory document when validation is concerned.

Elsewhere in the document there is a further statement: "A software requirements specification document should be created with a written definition of the software functions to be performed." It is not possible to validate software without predetermined and documented software requirements. Typical software requirements specify the following:

- all inputs that the software system will receive
- all outputs that the software system will produce
- all functions that the software system will perform
- all performance requirements that the software will meet, e.g., data throughput, reliability, timing, etc.
- the definition of all internal, external and user interfaces
- what constitutes an error and how errors should be handled
- the intended operating environment for the software, e.g., hardware platform, operating system, etc., (if this is a design constraint)
- all safety requirements, features or functions that will be implemented in software
- all ranges, limits, defaults and specific values that the software will accept.

There is a further requirement in the Quality System Regulation [Title 21, Code of Federal Regulations, Chapter 820.30(c)][6] that, "requires a mechanism for addressing incomplete, ambiguous, or conflicting requirements. Each software requirement documented in the software requirements specification should be evaluated for accuracy, completeness, consistency, testability, correctness, and clarity."

Furthermore, "A software requirements traceability analysis should be conducted to trace software requirements to system requirements (and vice versa). In addition, a software requirements interface analysis should be conducted, comparing the software requirements to hardware, user, operator and software interface requirements for accuracy, completeness, consistency, correctness, and clarity, and to assure that there are no external inconsistencies. In addition to any other analyses and documentation used to verify software requirements, one or more Formal Design Reviews (a.k.a. Formal Technical Reviews) should be conducted to confirm that requirements are fully specified and appropriate, before extensive software design efforts begin. Requirements can be approved and released incrementally, but care should be taken that interactions and interfaces

among software (and hardware) requirements are properly reviewed, analysed and controlled."

Remember when you read these draft guidelines that we are dealing with medical devices (including software itself) that could have life-threatening consequences as a result of an error. However, we can now see a regulatory body getting to grips with software in sufficient detail.

Worried about this approach? This is basic common sense and good computing practice that most of us forget when time pressures are on us. Remember also, where we started this article: discussing the nine US government software contracts in which 65% of the money was wasted because of poor, or misunderstood, user requirements specifications. Are you now convinced of the need for a URS?

## Summary

The bottom line is that a well-written user requirements specification will be time and resource well spent. The alternative is quite simple, just repeat the following words, 'but we don't actually work this way' every time you look at the functions of the delivered data system. Then practise kneeling and asking for money from your boss to pay for the enhancements you will need to get the system working.

In fact, Lewis Carroll, in Alice In Wonderland got it right:
"Where are you going?" said the cat.
"I don't know," said Alice.
"In that case," said the cat, "it does not matter as you will probably end up somewhere else."

Fine in fiction, but would you do this in practice with a computerized system? Of course you would!

## References

1. A.Mihandru, in Proceedings of the 3rd International Conference on Computers and Communications, IEEE Computer Society Press, Silver Spring, Maryland, USA, 1984.
2. IEEE Recommended Practice for Software Requirements Specifications, IEEE Standard 830-1993, in Software Engineering Standards, 1994 Edition, Institute of Electronic and Electrical Engineers, Piscataway, New Jersey, USA, 1994.
3. Appendix A: User Requirements Specifications, Good Automated Manufacturing Practices (GAMP), May 1996, International Society for Pharmaceutical Engineering, Tampa, Florida, USA.
4. Guideline for Process Validation, May 1987, Food and Drug Administration, Maryland, USA.
5. General Principles of Software Validation, Draft Guidance for Industry, June 1997, Food and Drug Administration.
6. Quality System Regulation, Title 21, Code of Federal Regulations, Chapter 820.30(c), 6 October 1996, effective 1 June 1997.

Bob McDowall is principal of McDowall Consulting, and is involved in the specification and validation of data-management systems and software applications. He is also visiting senior lecturer in the Department of Chemistry, University of Surrey, UK, and is a member of the Editorial Advisory Board of Scientific Data Management.