Magazine

What's fundamentally wrong? Improving our approach towards capturing value in requirements specification

Authors: Tom Gilb and Lindsey Brodie

About the authors:

Tom is the author of nine books, and hundreds of papers on these and related subjects. His latest book 'Competitive Engineering' is a substantial definition of requirements ideas. His ideas on requirements are the acknowledged basis for CMMI level 4 (quantification, as initially developed at IBM from 1980). Tom has guest lectured at universities all over UK, Europe, China, India, USA, Korea – and has been a keynote speaker at dozens of technical conferences internationally.
www.gilb.com, twitter: @imTomGilb

Lindsey Brodie is currently carrying out research on prioritization of stakeholder value, and teaching part-time at Middlesex University. She has an MSc in Information Systems Design from Kingston Polytechnic. Her first degree was Joint Honours Physics and Chemistry from King's College, London University. Lindsey worked in industry for many years, mainly for ICL. Initially, Lindsey worked on project teams on customer sites (including the Inland Revenue, Barclays Bank, and J. Sainsbury's) providing technical support and developing customised software for operations. From there, she progressed to product support of mainframe operating systems and data management software: databases, data dictionary and 4th generation applications. Having completed her Masters, she transferred to systems development - writing feasibility studies and user requirements specifications, before working in corporate IT strategy and business process re-engineering.

Lindsey has collaborated with Tom Gilb and edited his book, “Competitive Engineering”. She has also co-authored a student textbook, “Successful IT Projects” with Darren Dalcher (National Centre for Project Management). She is a member of the BCS and a Chartered IT Practitioner (CITP).

Advanced
Level

1
Magazine Number

Software Engineering
Section in the magazine

Abstract

We are all aware that many of our IT projects fail and disappoint: the poor state of requirements practice is frequently stated as a contributing factor. This article proposes a fundamental cause is that we think like programmers, not engineers and managers. We fail to concentrate on value delivery, and instead focus on functions, on use-cases and on code delivery. Our requirements specification practices fail to adequately address capturing value-related information. Compounding this problem, senior management is not taking its responsibility to make things better: managers are not effectively communicating about value and demanding value delivery. This article outlines some practical suggestions aimed at tackling these problems and improving the quality of requirements specification.

Keywords: Requirements; Value Delivery; Requirements Definition; Requirements Specification

Introduction

We know many of our IT projects fail and disappoint, and that the overall picture is not dramatically improving [1] [2]. We also know that the poor state of requirements practice is frequently stated as one of the contributing failure factors [3] [4]. However, maybe a more fundamental cause can be proposed? A cause, which to date has received little recognition, and that certainly fails to be addressed by many well known and widely taught methods. What is this fundamental cause? In a nutshell: that we think like programmers, and not as engineers and managers. In other words, we do not concentrate on value delivery, but instead focus on functions, on use cases and on code delivery. As a result, we pay too little attention to capturing value and value-related information in our requirements specifications. We fail to capture the information that allows us to adequately consider priorities, and engineer and manage stakeholder-valued solutions.

This article outlines some practical suggestions aimed at tackling these problems and improving the quality of requirements specification. It focuses on ‘raising the bar’ for communicating about value within our requirements. Of course, there is much still to be learnt about specifying value, but we can make a start – and achieve substantial improvement in IT project delivery – by applying what is already known to be good practice.

Note there is little that is new in what follows, and much of what is said can be simply regarded as commonsense. However, since IT projects continue not to grasp the significance of the approach advocated, and as there are people who have yet to encounter this way of thinking, it is worth repeating!

Definition of Value

The whole point of a project is achieving 'realized value' (also known as 'benefits'), for the stakeholders: it is not the defined functionality, and not the user stories that actually count. Value can be defined as 'the benefit we think we get from something' [5, page 435]. See Figure 1.

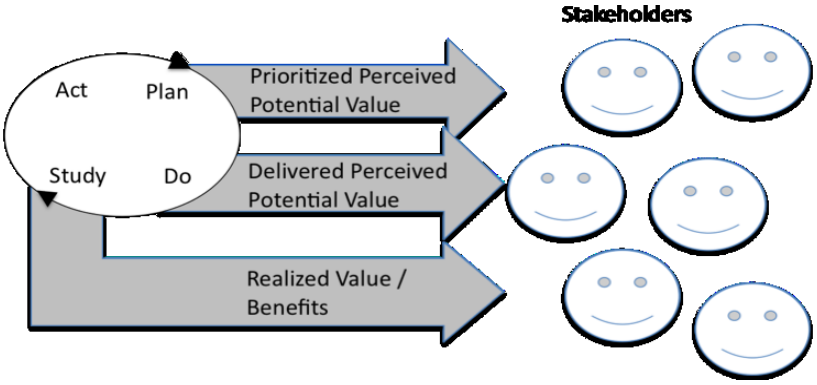

Figure 1 Value can be delivered gradually to stakeholders. Different stakeholders will perceive different value.

Notice the subtle distinction between initially perceived value ('I think that would be useful'), and realized value: effective and factual value ('this was in practice more valuable than we thought it would be, because …'). Realized value has dependencies on the stakeholders actually utilizing a project's deliverables.

The issue with much of the conventional requirements thinking is that it is not closely enough coupled with 'value'. IT business analysts frequently fail to gather the information supporting a more precise understanding and/or the calculation of value. Moreover, the business people when stating their requirements frequently fail to justify them using value. The danger if requirements are not closely tied to value is that we lack the basic information allowing us to engineer and prioritize implementation to achieve value delivery, and we risk failure to deliver the required expected value, even if the 'requirements' are satisfied.

It is worth pointing out that 'value' is multi-dimensional. A given requirement can have financial value, environmental value, competitive advantage value, architectural value, as well as many other dimensions of value. Certainly value requires much more explicit definition than the priority groups used by MoSCoW ('Must Have', 'Should Have', 'Could Have', and 'Would like to Have/Won't Have This Time') [6] or by the Planning Game ('Essential', 'Less Essential' and 'Nice To Have') [7] for prioritizing requirements. Further, for an IT project, engineering 'value' also involves consideration of not just the requirements, but also the optional designs and the resources available: tradeoffs are needed. However, these are topics for future articles, this article focuses on the initial improvements needed in requirements specification to start to move towards value thinking.

Definition of Requirement

Do we all have a shared notion of what a 'requirement' is? This is another of our problems. Everybody has an opinion, and many of the opinions about the meaning of the concept 'requirement' are at variance: few of the popular definitions are correct or useful - especially when you consider the concept of 'value' alongside them. We have decided to define a requirement as a "stakeholder-valued end state". You possibly will not accept, or use this definition yet, but we have chosen it to emphasize the 'point' of IT systems engineering.

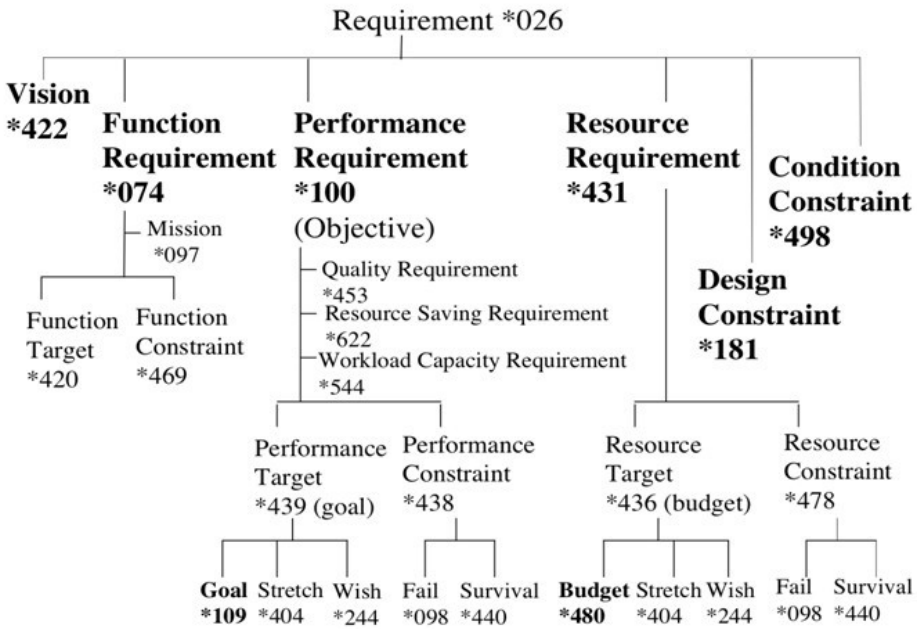

Figure 2 Example of Planguage requirements concepts

In previous work, we have identified, and defined a large number of requirement concepts [5, see Glossary, pages 321-438]. A sample of these concepts is given in Figure 2. You can use these concepts and the notion of a "stakeholder-valued end state" to re-examine your current requirements specifications. In the rest of this article, we provide more detailed discussion about some of the key points (the "key principles") you should consider.

The Key Principles

The key principles are summarized in Figure 3. Let's now examine these principles in more detail. Note, unless otherwise specified, further details on all aspects of Planguage (a planning language developed by one of the authors, Tom Gilb) can be found in [5].

Ten Key Principles for Successful Requirements

1. Understand the top level critical objectives
2. Think stakeholders: not just users and customers!
3. Focus on the required system quality, not just its functionality
4. Quantify quality requirements as a basis for software engineering
5. Don't mix ends and means
6. Capture explicit information about value
7. Ensure there is 'rich specification': requirement specifications need far more information than the requirement itself!
8. Carry out specification quality control (SQC)
9. Consider the total lifecycle and apply systems-thinking - not just a focus on software
10. Recognize that requirements change: use feedback and update requirements as necessary

Figure 3 Ten Key Principles for Successful Requirements

Principle 1. Understand the top-level critical objectives

The 'worst requirement sin of all' is found in almost all the IT projects we look at, and this applies internationally. Time and again, the high-level requirements – also known as the top-level critical objectives (the ones that fund the project), are vaguely stated, and ignored by the project team. Such requirements frequently look like the example given in Figure 4 (which has been slightly edited to retain anonymity). These requirements are for a real project that ran for eight years and cost over 100 million US dollars. The project failed to deliver any of them. However, the main problem is that these are not top-level critical objectives: they fail to explain in sufficient detail what the business is trying to achieve: there are no real pointers to indicate the business aims and priorities. There are additional problems as well that will be discussed further later (such as lack of quantification, mixing optional designs into the requirements, and insufficient background description).

Example of Initial Weak Top-Level Critical Objectives

1. Central to the corporation's business strategy is to be the world's premier integrated <domain> service provider
2. Will provide a much more efficient user experience
3. Dramatically scale back the time frequently needed after the last data is acquired to time align, depth correct, splice, merge, recomputed and/or do whatever else is needed to generate the desired products
4. Make the system much easier to understand and use than has been the case with the previous system
5. A primary goal is to provide a much more productive system development environment then was previously the case
6. Will provide a richer set of functionality for supporting next generation logging tools and applications
7. Robustness is an essential system requirement
8. Major improvements in data quality over current practices

Figure 4 Example of Initial Weak Top Level Critical Objectives

Management at the CEO, CTO and CIO level did not take the trouble to clarify these critical objectives. In fact, the CIO told me that the CEO actively rejected the idea of clarification! So management lost control of the project at the very beginning. Further, none of the technical 'experts' reacted to the situation. They happily spent $100 million on all the many suggested architecture solutions that were mixed in with the objectives.

It actually took less than an hour to rewrite one of these objectives, "Robustness", so that it was clear, measurable, and quantified (see later). So in one day's work the project could have clarified the objectives, and perhaps avoided some of the eight years of wasted time and effort.

Principle 2. Think stakeholders: not just users and customers!

Too many requirements specifications limit their scope to being too narrowly focused on user or customer needs. The broader area of stakeholder needs and values should be considered, where a 'stakeholder' is anyone or anything that has an interest in the system [5, page 420]. It is not just the users and customers that must be considered: IT development, IT maintenance, senior management, operational management, regulators, government, as well as other stakeholders can matter. The different stakeholders will have different viewpoints on the requirements and their associated value. Further, the stakeholders will be "experts" in different areas of the requirements. These different viewpoints will potentially lead to differences in opinion over the implementation priorities.

Principle 3. Focus on the required system quality, not just its functionality

Far too much attention is paid to what the system must do (function) and far too little attention is given to how well it should do it (qualities). Many requirements specifications consist of detailed explanation of the functionality with only brief description of the required system quality. This is in spite of the fact that quality improvements tend to be the major drivers for new projects.

In contrast, here's an example, the Confirmit case study [8], where the focus of the project was not on functionality, but on driving up the system quality. By focusing on the "Usability" and "Performance" quality requirements the project achieved a great deal! See Table 1.

Description of requirement/work task	**Past**	**Current Status**
Usability.Productivity: Time for the system to generate a survey	7200 sec	15 sec
Usability.Productivity: Time to set up a typical market research report	65 min	20 min
Usability.Productivity: Time to grant a set of end-users access to a report set and distribute report login information	80 min	5 min
Usability.Intuitiveness: The time in minutes it takes a medium-experienced programmer to define a complete and correct data transfer definition with Confirmit Web Services without any user documentation or any other aid	15 min	5 min
Performance.Runtime.Concurrency: Maximum number of simultaneous respondents executing a survey with a click rate of 20 sec and a response time < 500ms given a defined [Survey Complexity] and a defined [Server Configuration, Typical]	250 users	6000

Table 1 Extract from Confirmit Case Study [8]

By system quality we mean all the "-ilities" and other qualities that a system can express. Some system developers limit system quality to referring to bug levels in code. However, a broader definition should be used. System qualities include availability, usability, portability, and any other quality that a stakeholder is interested in, like intuitiveness or robustness. See Figure 5, which shows a set of quality requirements. It also shows the notion that resources are "input" or used by a function, which in turn "outputs" or expresses system qualities. Sometimes the system qualities are mis-termed "non-functional requirements (NFRs)", but as can be seen in this figure, the system qualities are completely linked to the system functionality. In fact, different parts of the system functionality are likely to require different system qualities.

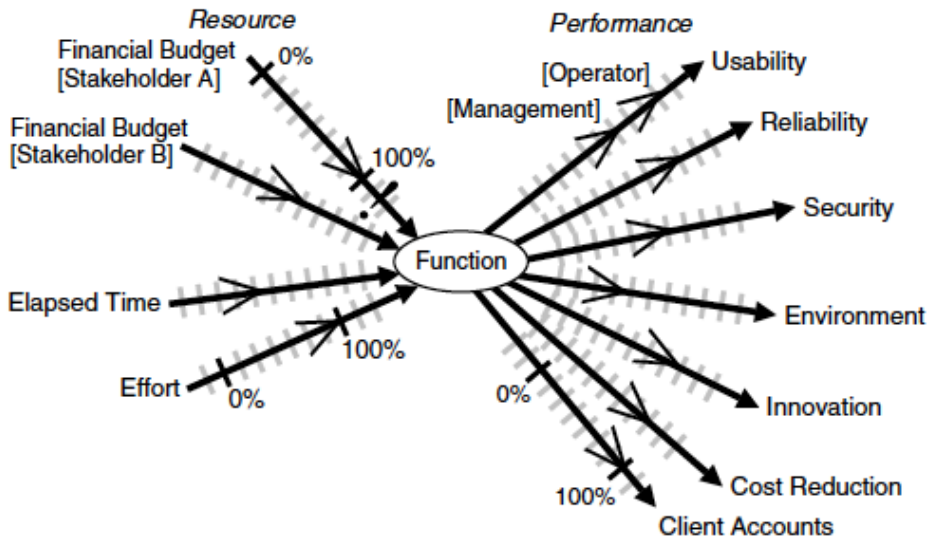

Figure 5 A way of visualizing qualities in relation to function and cost. Qualities and costs are scalar variables, so we can define scales of measure in order to discuss them numerically. The arrows on the scale arrows represent interesting points, such as the requirement levels. The requirement is not 'security' as such, but a defined, and testable degree of security [5, page 163]

Principle 4. Quantify quality requirements as a basis for software engineering

Frequently we fail to practice "software engineering" in the sense of real engineering as described by engineering professors, like Koen [9]. All too often quality requirements specifications consist merely of words. No numbers, just nice sounding words; good enough to fool managers into spending millions for nothing (for example, "a much more efficient user experience").

We seem to almost totally avoid the practice of quantifying qualities. Yet we need quantification in order to make the quality requirements clearly understood, and also to lay the basis for measuring and tracking our progress in improvement towards meeting them. Further, it is the quantification that is the key to a better understanding of cost and value – different levels of quality have different associated cost and value.

The key idea for quantification is to define, or reuse a definition, of a scale of measure. For example, for a quality "Intuitiveness", a sub-component of "Usability":

Usability.Intuitiveness:
Type: Marketing Product Quality Requirement.
Ambition: Any potential user, any age, can immediately discover and correctly use all functions of the product, without training, help from friends, or external documentation.
Scale: % chance that defined [User] can successfully complete defined [Tasks] <immediately> with no external help.
Meter: Consumer reports tests all tasks for all defined user types, and gives public report.
Goal [Market = USA, User = Seniors, Product = New Version, Task = Photo Tasks Set, When = 2012]**:** 80% ±10% <- Draft Marketing Plan.

Figure 6 A simple example of quantifying a quality requirement, 'Intuitiveness'.

To give some explanation of the key quantification features in Figure 6:

1. Ambition is a high-level summary of the requirement. One that is easy to agree to, and understand roughly.
2. Scale is the formal definition of our chosen scale of measure. The parameters [User] and [Task] allow us to generalize here, while becoming more specific in detail below (see later). They also encourage and permit the reuse of the Scale, as a sort of 'pattern'.
3. Meter provides a defined measuring process. There can be more than one for different occasions.
4. Goal is one of many possible requirement levels (see earlier detail in Figure 2 for some others: Stretch, Wish, Fail and Survival). We are defining a stakeholder-valued future state (for example: 80% ± 10%).

One *stakeholder* is 'USA Seniors'. The *future* is 2012. The requirement level type, Goal, is defined as a very high priority, budgeted promise of delivery. It is of higher priority than a Stretch or Wish level. Note other priorities may conflict and prevent this particular requirement from being delivered in practice.

If you know the *conventional* state of requirements methods, then you will now, from this example alone, begin to appreciate the difference proposed by such quantification - especially for *quality* requirements. IT projects already quantify time, cost,, response time, burn rate, and bug density – but there is much *more to achieve system engineering*!

Here is another example of quantification (see Figure 7). It is the initial stage of the rewrite of Robustness from the Figure 4 example. First we determined that Robustness is complex and composed of many different attributes, such as Testability.

Robustness:
Type: *Complex* Product Quality Requirement.
Includes: {Software Downtime, Restore Speed, Testability, Fault Prevention Capability, Fault Isolation Capability, Fault Analysis Capability, Hardware Debugging Capability}.

Figure 7 Definition of a complex quality requirement, Robustness

Then we defined Testability in more detail (see Figure 8).

Testability:
Type: Software Quality Requirement.
Version: Oct 20, 2006.
Status: Draft.
Stakeholder: {Operator, Tester}.
Ambition: Rapid duration automatic testing of <critical complex tests> with extreme operator setup and initiation.
Scale: The duration of a defined [Volume] of testing or a defined [Type of Testing] by a defined [Skill Level] of system operator under defined [Operating Conditions].
Goal [All Customer Use, Volume = 1,000,000 data items, Type of Testing = WireXXXX vs. DXX, Skill Level = First Time Novice, Operating Conditions = Field]**:** < 10 minutes.
Design: Tool simulators, reverse cracking tool, generation of simulated telemetry frames entirely in software, application specific sophistication for drilling – recorded mode simulation by playing back the dump file, application test harness console <- 6.2.1 HFS.

Figure 8 Quantitative definition of Testability, an attribute of Robustness

Note this example shows the notion of there being different levels of requirements. Principle 1 also has relevance here as it is concerned with top-level objectives (requirements). The different levels that can be identified include: corporate requirements, the top-level critical few project or product requirements, system requirements and software requirements. We need to clearly document the level and the interactions amongst these requirements.

An additional notion is that of 'sets of requirements'. Any given stakeholder is likely to have a set of requirements rather than just an isolated single requirement. In fact, achieving value could depend on meeting an entire set of requirements.

Principle 5. Don't mix ends and means

"Perfection of means and confusion of ends seem to characterize our age." Albert Einstein. 1879-1955

The problem of confusing ends and means is clearly an old one, and deeply rooted. We specify a solution, design and/or architecture, instead of what we really value – our real requirement. There are explanatory reasons for this – for example solutions are more concrete, and what we want (qualities) are more abstract for us (because we have not yet learned to make them measurable).

The problems occur when we do confuse them: if we do specify the means, and not our true ends. As the saying goes: "Be careful what you ask for, you might just get it" (unknown source). The problems include:

- You might not get what you *really* want
- The solution you have specified might cost *too much* or have bad *side effects*, even if you do get what you want
- There may be *much better* solutions you don't know about yet.

So how to we find the 'right requirement', the 'real requirement' [10] that is being 'masked' by the solution? *Assume* that there probably is a better formulation, which is a more accurate expression of our real values and needs. Search for it by asking 'Why?' Why do I want X, it is because I really want Y, and assume I will get it through X. But, then why do I want Y? Because I really want Z and assume that is the best way to get X. Continue the process until it seems reasonable to stop. This is a slight variation on the '5 Whys' technique [11], which is normally used to identify root causes of problems (rather than high-level requirements).

Assume that our stakeholders will usually state their values in terms of some perceived means to get what they really value. Help them to identify (The 5 Whys?) and to acknowledge what they really want, and make that the 'official' requirement. Don't insult them by telling them that they don't know what they want. But explain that you will help them more-certainly get what they more deeply want, with better and cheaper solutions, perhaps new technology, if they will go through the '5 Whys?' process with you. See Figure 9.

Why do you require a 'password'? For Security!
What kind of security do you want? Against stolen information.
What level of strength of security against stolen information are you willing to pay for? At least a 99% chance that hackers cannot break in within 1 hour of trying! Whatever that level costs up to €1 million.
So that is your real requirement? Yep.

Can we make that the official requirement, and leave the security design to both our security experts, and leave it to proof by measurement to decide what is really the right design? Of course! The aim being that whatever technology we choose, it gets you the 99%?
Sure, thanks for helping me articulate that!

Figure 9 Example of the requirement, not the design feature, being the real requirement

Note that this separation of designs from the requirements does not mean that you ignore the solutions/designs/architecture when software engineering. It is just that you must separate your requirements - including any mandatory means - from any optional means. The key thing is to understand what is optional so that you consider alternative solutions. See Figure 10, which shows two alternative solutions: Design A with Designs B and C, or Design A with Design D. Assuming that say, Design B was mandatory, could distort your project planning.

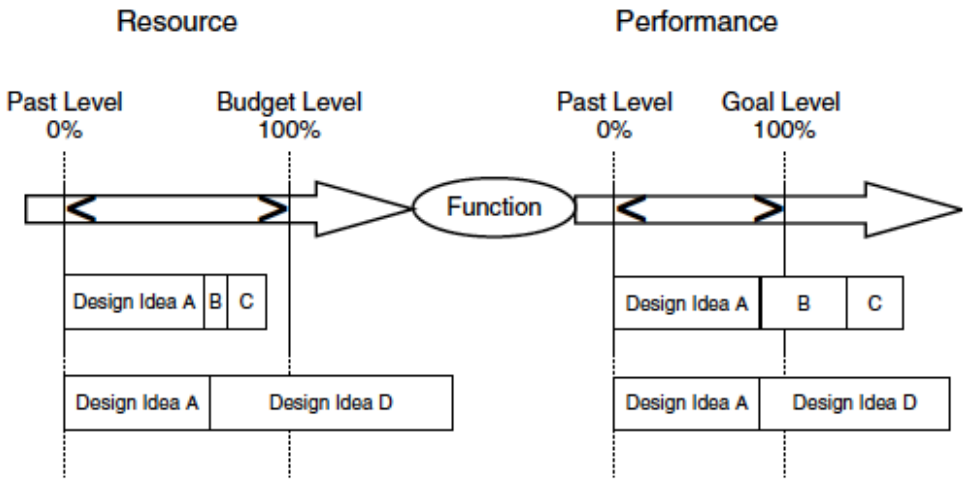

Figure 10 A graphical way of understanding performance attributes (which include all qualities) in relation to function, design and resources. Design ideas cost some resources, and design ideas deliver performance (including system qualities) for given functions.

Principle 6. Capture explicit information about value

How can we articulate and document notions of value in a requirement specification? See the example for Intuitiveness, a component quality of Usability, given in Figure 11, which expands on Figure 6.

Usability.Intuitiveness:
Type: Marketing Product Requirement.
Stakeholders: {Marketing Director, Support Manager, Training Center}.
Impacts: {Product Sales, Support Costs, Training Effort, Documentation Design}.
Supports: Corporate Quality Policy 2.3.
Ambition: Any potential user, any age, can immediately discover and correctly use all functions of the product, without training, help from friends, or external documentation.
Scale: % chance that a defined [User] can successfully complete the defined [Tasks] <immediately>, with no external help.
Meter: Consumer Reports tests all tasks for all defined user types, and gives public report.
---------- Analysis ----------
Trend [Market = Asia, User = {Teenager, Early Adopters}, Product = Main Competitor, Projection = 2013]**:** 95%±3% <- Market Analysis.

Past [Market = USA, User = Seniors, Product = Old Version, Task = Photo Tasks Set, When = 2010]**:** 70% ±10% <- Our Labs Measures.
Record [Market = Finland, User = {Android Mobile Phone, Teenagers}, Task = Phone+SMS Task Set, Record Set = January 2010]**:** 98% ±1% <- Secret Report.
---------- Our Product Plans ----------
Goal [Market = USA, User = Seniors, Product = New Version, Task = Photo Tasks Set, When = 2012]**:** 80% ±10% <- Draft Marketing Plan.
Value [Market =USA, User = Seniors, Product = New Version, Task = Photo Tasks Set, Time Period = 2012]**:** 2M USD.
Tolerable [Market = Asia, User = {Teenager, Early Adopters}, Product = Our New Version, Deadline = 2013]**:** 97%±3% <- Marketing Director Speech.
Fail [Market = Finland, User = {Android Mobile Phone, Teenagers}, Task = Phone+SMS Task Set, Product Release 9.0]**:** Less Than 95%.
Value [Market = Finland, User = {Android Mobile Phone, Teenagers}, Task = Phone+SMS Task Set, Time Period = 2013]**:** 30K USD.

Figure 11 A fictitious Planguage example, designed to display ways of making the value of a requirement clear

For brevity, a detailed explanation is not given here. Hopefully, the Planguage specification is reasonably understandable without detailed explanation. For example, the Goal statement (80%) specifies which market ("USA") and users ("Seniors") it is intended for, which set of tasks are valued (the "Photo Tasks Set"), and when it would be valuable to get it delivered ("2012"). This 'qualifier' information in all the statements, helps document where, who, what, and when the quality level applies. The additional Value parameter specifies the perceived value of achieving 100% of the requirement. Of course, more could be said about value and its specification, this is merely a 'wake-up call' that explicit value needs to be captured within requirements. It is better than the more common specifications of the Usability requirement, that we often see, such as: "The product will be more user-friendly, using Windows".

So who is going to make these value statements in requirements specifications? I don't expect developers to care much about value statements. Their job is to deliver the requirement levels that someone else has determined are valued. Deciding what sets of requirements are valuable is a Product Owner (Scrum) or Marketing Management function. Certainly, the IT staff should only determine the value related to IT stakeholder requirements!

Principle 7. Ensure there is 'rich specification': requirement specifications need far more information than the requirement itself!

Far too much emphasis is often placed on the requirement itself; and far too little concurrent information is gathered about its background, for example: who wants this requirement and when? The requirement itself might be less than 10% of a complete requirement specification that includes the background information. It should be a corporate standard to specify this related background information, and to ensure it is intimately and immediately tied into the requirement itself.

Such background information is useful related information, but is not central (core) to the implementation, and nor is it commentary. The central information includes: Scale, Meter, Goal, Definition and Constraint.

Background specification includes: benchmarks {Past, Record, Trend}, Owner, Version, Stakeholders, Gist (brief description), Ambition, Impacts, and Supports. The rationale for background information is as follows:

- To help judge the value of the requirement
- To help prioritize the requirement
- To help understand the risks associated with the requirement
- To help present the requirement in more or less detail for various audiences and different purposes
- To give us help when updating a requirement
- To synchronize the relationships between different but related levels of the requirements
- To assist in quality control of the requirements
- To improve the clarity of the requirement.

Commentary is any detail that probably will not have any economic, quality or effort consequences if it is incorrect, for example, notes and comments.

See Figure 12 for an example, which illustrates the help given by background information regarding risks.

Reliability:
Type: Performance Quality.
Owner: Quality Director. **Author:** John Engineer.
Stakeholders: {Users, Shops, Repair Centers}.
Scale: Mean Time Between Failure.
Goal [Users]: 20,000 hours <- Customer Survey, 2004.
Rationale: Anything less would be uncompetitive.
Assumption: Our main competitor does not improve more than 10%.
Issues: New competitors might appear.
Risks: The technology costs to reach this level might be excessive.
Design Suggestion: Triple redundant software and database system.
Goal [Shops]: 30,000 hours <- Quality Director.
Rationale: Customer contract specification.
Assumption: This is technically possible today.
Issues: The necessary technology might cause undesired schedule delays.
Risks: The customer might merge with a competitor chain and leave us to foot the costs for the component parts that they might no longer require.
Design Suggestion: Simplification and reuse of known components.

Figure 12 A requirement specification can be embellished with many background specifications that will help us to understand risks associated with one or more elements of the requirement specification [12].

Background information must not be scattered around in different documents and meeting notes. It needs to be directly integrated into a sole master reusable requirement specification object. Otherwise it will not be available when it is needed: it will not be updated, or shown to be inconsistent with emerging improvements in the requirement specification.

See Figure 13 for a requirement template for function specification [5, page 106], which hints at the richness possible for background information.

TEMPLATE FOR FUNCTION SPECIFICATION <with hints>

Tag: <Tag name for the function>.
Type: <{Function Specification, Function (Target) Requirement, Function Constraint}>.
========== **Basic Information** ==========
Version: <Date or other version number>.
Status: <{Draft, SQC Exited, Approved, Rejected}>.
Quality Level: <Maximum remaining major defects/page, sample size, date>.

Owner: <Name the role/email/person responsible for changes and updates to this specification>.
Stakeholders: <Name any stakeholders with an interest in this specification>.
Gist: <Give a 5 to 20 word summary of the nature of this function>.
Description: <Give a detailed, unambiguous description of the function, or a tag reference to someplace where it is detailed. Remember to include definitions of any local terms>.
========== **Relationships** ==========
Supra-functions: <List tag of function/mission, which this function is a part of. A hierarchy of tags, such as A.B.C, is even more illuminating. Note: an alternative way of expressing supra-function is to use Is Part Of>.
Sub-functions: <List the tags of any immediate sub-functions (that is, the next level down), of this function. Note: alternative ways of expressing sub-functions are Includes and Consists Of>.
Is Impacted By: <List the tags of any design ideas or Evo steps delivering, or capable of delivering, this function. The actual function is NOT modified by the design idea, but its presence in the system is, or can be, altered in some way. This is an Impact Estimation table relationship>.
Linked To: <List names or tags of any other system specifications, which this one is related to intimately, in addition to the above specified hierarchical function relations and IE-related links. Note: an alternative way is to express such a relationship is to use Supports or Is Supported By, as appropriate>.
========== **Measurement** ==========
Test: <Refer to tags of any test plan or/and test cases, which deal with this function>.
========== **Priority and Risk Management** ==========
Rationale: < Justify the existence of this function. Why is this function necessary? >.
Value: <Name [Stakeholder, time, place, event>]: <Quantify, or express in words, the value claimed as a result of delivering the requirement>.
Assumptions: <Specify, or refer to tags of any assumptions in connection with this function, which could cause problems if they were not true, or later became invalid>.
Dependencies: <Using text or tags, name anything, which is dependent on this function in any significant way, or which this function itself, is dependent on in any significant way>.
Risks: <List or refer to tags of anything, which could cause malfunction, delay, or negative impacts on plans, requirements and expected results>.
Priority: <Name, using tags, any system elements, which this function can clearly be done *after* or must clearly be done *before*. Give any relevant reasons>.
Issues: <State any known issues>.
========== **Specific Budgets** ==========
Financial Budget: <Refer to the allocated money for planning and implementation (which includes test) of this function>.

Figure 13 A template for function specification [5, page 106]

Principle 8. Carry out specification quality control (SQC)

There is far too little quality control of requirements against relevant standards. All requirements specifications ought to pass their quality control checks before they are released for use by the next processes. Initial quality control of requirements specification, where there has been no previous use of specification quality control (SQC) (also known as Inspection), using three simple quality-checking rules ('unambiguous to readers', 'testable' and 'no optional designs present'), typically identifies 80 to 200+ words per 300 words of requirement text as ambiguous or unclear to intended readers! [13]

Principle 9. Consider the total lifecycle and apply systems-thinking - not just a focus on software

If we don't consider the total lifecycle of the system, we risk failing to think about all the things that are necessary prerequisites to actually delivering full value to real stakeholders on time. For example, if we want better maintainability then it has to be designed into the system. If we are really engineering costs, then we need to think about the total operational costs over time. This is much more than just considering the programming aspects.

You must take into account the nature of the system: an exploratory web application doesn't need to same level of software engineering as a real-time banking system!

Principle 10. Recognise that requirements change: use feedback and update requirements as necessary

Ideally requirements must be developed based on on-going feedback from stakeholders, as to their real value. System development methods, such as the agile methods, enable this to occur. Stakeholders can give feedback about their perception of value, based on the realities of actually using the system. The requirements must be evolved based on this realistic experience. The whole process is a 'Plan Do Study Act' Shewhart cyclical learning process involving many complex factors, including factors from outside the system, such as politics, law, international differences, economics, and technology change.

Attempts to fix the requirements in advance of feedback, are typically wasted energy (unless the requirements are completely known upfront, which might be the case in a straightforward system rewrite with no system changes). Committing to fixed requirements specifications in contracts is not realistic.

Who or What Will Change Things?

Everybody talks about requirements, but few people seem to be making progress to enhance the quality of their requirements specifications and improve support for software engineering. Yes, there are internationally competitive businesses, like HP and Intel that have long since improved their practices because of their competitive nature and necessity [8, 14]. But they are very different from the majority of organizations building software. The vast majority of IT systems development teams we encounter are not highly motivated to learn or practice first class requirements (or anything else!). Neither the managers nor the systems developers seem strongly motivated to improve. The reason is that they get by with, and even get well paid for, failed projects.

The universities certainly do not train IT/computer science students well in requirements, and the business schools also certainly do not train managers about such matters [15]. The fashion now seems to be to learn oversimplified methods, and/or methods prescribed by some certification or standardization body. Perhaps insurance companies and lawmakers might demand better industry practices, but I fear that even that would be corrupted in practice if history is any guide (for example, think of CMMI and the various organization certified as being at Level 5).

Summary

Current requirements specification practice is often woefully inadequate for today's critical and complex systems. Yet we do know a considerable amount (Not all!) about good practice. The main question is whether your 'requirements' actually capture the true breadth of information that is needed to make a start on engineering value for your stakeholders.

Here are some specific questions for you to ask about your current IT project's requirements specification:

- Do you have a list of top-level critical objectives?
- Do you consider multiple stakeholder viewpoints?
- Do you know the expected stakeholder value to be delivered?
- Have you quantified your top five quality attributes? Are they are testable? What are the current levels for these quality attributes?
- Are there any optional designs in your requirements?
- Can you state the source of each of your requirements?
- What is the quality level of your requirements documentation? That is, the number of major defects remaining per page?

- When are you planning to deliver stakeholder value? To which stakeholders?

If you can't answer these questions with the 'right' answers, then you have work to do! And you might also better understand why your IT project is drifting from delivering its requirements. The good news is that the approach outlined in this article should allow you to focus rapidly on what really matters to your stakeholders: value delivery.

References

1. Thomas Carper, Report Card to the Senate Hearing "Off-Line and Off-Budget: The Dismal State of Federal Information Technology Planing", July 31, 2008. See http://uscpt.net/CPT_InTheNews.aspx [Last Accessed: August 2010].
2. The Standish Group, "Chaos Summary 2009", 2009. See http://www.standishgroup.com/newsroom/chaos_2009.php [Last Accessed: August 2010].
3. John McManus and Trevor Wood-Harper, "A Study in Project Failure", June 2008. See http://www.bcs.org/server.php?show=ConWebDoc.19584 [Last Accessed: August 2010].
4. David Yardley, Successful IT Project Delivery, Addison-Wesley, 2002. ISBN 0201756064.
5. Tom Gilb, "Competitive Engineering: A Handbook for Systems Engineering, Requirements Engineering, and Software Engineering using Planguage", Elsevier Butterworth-Heinemann, 2005.
6. Jennifer Stapleton (Editor), DSDM: Business Focused Development (2nd Edition), Addison Wesley, 2003. ISBN 0321112245. First edition published in 1997.
7. Mike Cohn, User Stories Applied: For Agile Software Development, Addison Wesley, 2004. ISBN 0321205685.
8. Trond Johansen and Tom Gilb, From Waterfall to Evolutionary Development (Evo): How we created faster, more user-friendly, more productive software products for a multi-national market, Proceedings of INCOSE, 2005. See http://www.gilb.com/tiki-download_file.php?fileId=32
9. Dr. Billy Vaughn Koen, "Discussion of the Method: Conducting the Engineer's Approach to Problem Solving", Oxford University Press, 2003.
10. Tom Gilb, Real Requirements, see http://www.gilb.com/tiki-download_file.php?fileId=28
11. Taiichi Ohno, "Toyota production system: beyond large-scale production", Productivity Press, 1988.
12. Tom Gilb, "Rich Requirement Specs: The use of Planguage to clarify requirements", see http://www.gilb.com/tiki-download_file.php?fileId=44
13. Tom Gilb, Agile Specification Quality Control, Testing Experience, March 2009. Download from www.testingexperience.com/testing experience01_08.pdf [Last Accessed: August 2010].
14. Top Level Objectives: A slide collection of case studies. See http://www.gilb.com/tiki-download_file.php?fileId=180
15. Kenneth Hopper and William Hopper, "The Puritan Gift", I. B. Taurus and Co. Ltd., 2007.