VFQ Value Flow Quality

Requirements

by emergn

This publication forms part of Value, Flow, Quality® Education. Details of this and other Emergn courses can be obtained from Emergn, 20 Harcourt Street, Dublin, D02 H364, Ireland or Emergn, 190 High St, Floor 4, Boston, MA 02110, USA.

Alternatively, you may visit the Emergn website at https://www.emergn.com/education where you can learn more about the range of courses on offer.

To purchase Emergn's Value, Flow, Quality® courseware, contact us for a brochure and more details - tel. +44 (0)808 189 2043; email valueflowquality@emergn.com

First published 2012, revised 2022 - printed 14 January 2022 (version 2.1)

Emergn Ltd.
20 Harcourt Street
Dublin, D02 H364
Ireland

Emergn Inc.
190 High St, Floor 4
Boston, MA 02110
USA

Printed and bound in the United Kingdom by Apple Capital Print.

CONTENTS

INTRODUCTION

How important are requirements in delivering projects?

> "With about half of all software errors originating in the requirements activity, it is clear that a better understanding of the problem is needed … Getting the requirements right is crucial."
> **Suzanne Robertson,** Mastering the Requirement Process

> "The importance of complete, consistent and well documented software requirements is difficult to overstate."
> **Marvin Early,** Relating Software Requirements to Software Design

> "Get everything you can written in paper, drawn out, sketched on a cocktail napkin, signed, approved, sealed and delivered before you even start the user interface construction."
> **Chris Culos,** Studio Symposium

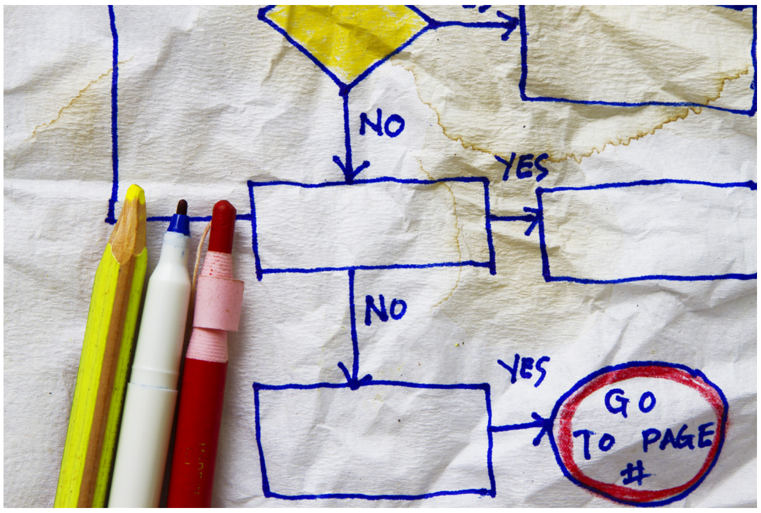


Figure 1. Cocktail napkins get crowded pretty quickly

> "More and more development organizations realize they cannot succeed unless they get the software requirements right."
> **Karl Wiegers,** More About Software Requirements

That's just a brief selection of how most of the leading authors in print today think about requirements and their essential role in software development projects. The same can be said in non-software projects and all forms of product design and development. Even explicitly Agile authors, who may have slightly different ideas on how requirements should be gathered and formulated, are clear on their importance to 'deliver better solutions to their users and stakeholders, and thereby achieve the personal and business benefits that success engenders' (Dean

Leffingwell, Agile Software Requirements). All of which sounds fairly important.

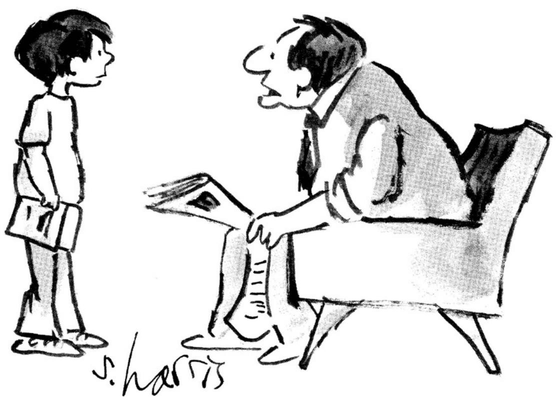
No wonder we worry about requirements. A study on project failure by the Project Management Institute found that for projects that were cancelled, requirements were at fault in 32% of all cases. In another study, authors Dr. John McManus and Dr. Trevor Wood-Harper also felt requirements were a key source of project failure, condemning 'the lack of due diligence at the requirements phase'.

32%

When projects fail, inaccurate requirements gathering is cited as the primary reason 32% of the time

(Project Management Institute, Inc. Pulse of the Profession March 2013 - www.pmi.org/pulse)

Perhaps unsurprisingly given requirements' seemingly crucial and fundamental importance, an enormous industry has sprung up devoted to the business of creating and managing them – you may be part of it yourself. The role of Business Analyst (or Requirements Engineer or System Analyst) is relatively new, even for a new industry, but it is growing fast. Research by Money Magazine and Salary.com, ranked "Computer IT Analyst" to be the 7th best job in America with an average salary of $83,500, with 67,300 annual job openings and a 10-year growth outlook of 36%. The International Institute of Business Analysis (IIBA) was formed in 2003, by 2011 it had 16,000 members.



"Sure you can become a systems analyst if you want to—but tell Daddy, what *is* a systems analyst?"

Figure 2. Selecting your IT career

Your role may be changing as you interact with software development teams that are following an Agile development process where little analysis is done up front and much planning and development happens as a result of feedback. This session aims to explain how requirements fit into a development process and how the whole team can become better at eliciting and articulating customer needs, and translating that information into useful, valuable solutions.

So it may not make us too popular when we tell you that some of this session will be devoted to explaining why it is that we think the very way we approach requirements is flawed. Even worse, many of the processes and practices developed in recent years to try to make the management of requirements better have ended up creating or reinforcing the very problems they set out to remove.

Agile and Lean practitioners tend to get it just as wrong as Waterfall or other types of traditional managers. Instead of using Agile's improvements in iterative design and feedback, they end up with a linear relationship in which over-heavy investment in requirements planning separates design and delivery. Dave Snowden, an insightful consultant, memorably wrote, 'Drawing a linear process as a circle does not make it any less linear...'

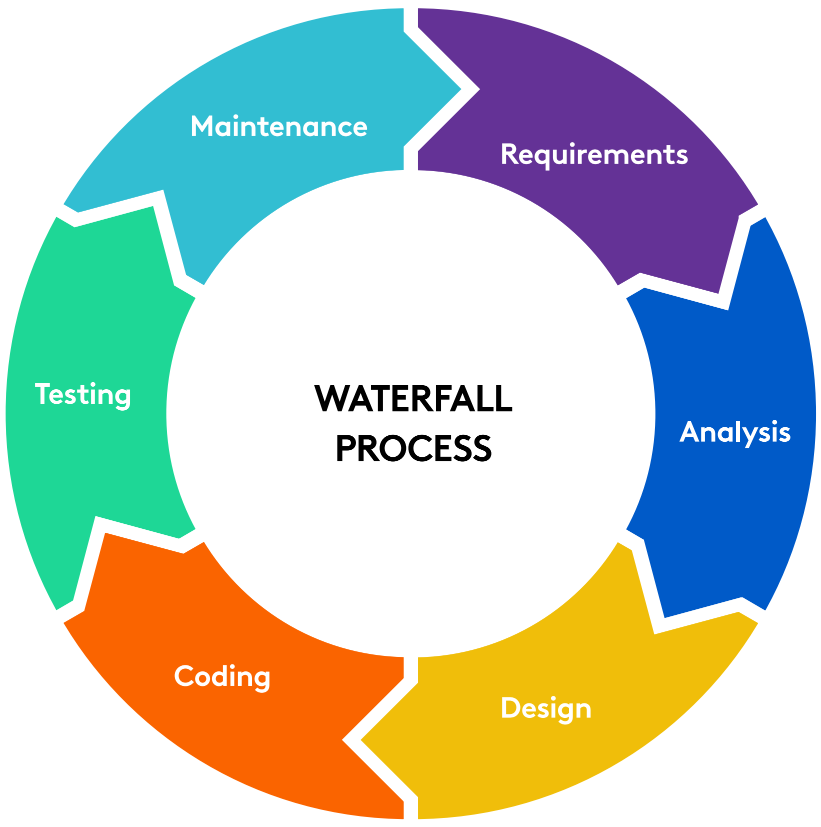


Figure 3. Waterfall - doesn't matter how you draw it, it's still not Agile

By the end of this session, you will be able to:

1. Identify how the term requirement is used in projects today.
2. Explore problems caused by the conventional approach to requirements, particularly the impact of separating design from delivery.
3. Appreciate the consequences of seeing requirements as information.
4. Judge how Agile practices differ in their conception of a requirement.
5. Understand and apply differing techniques for working with requirements. Select when each is most appropriate and appraise their advantages and disadvantages:
    - User stories
    - Decomposition
    - Feature injection
    - Critical requirements
    - Elaboration
    - Story mapping
    - Confirmation including acceptance criteria and specification by example
    - Supplementary tools including Agile modelling and use cases
6. Recognise the importance of non-functional requirements. Discriminate when they are used improperly and distinguish how to use them better in the context of your project.

1 WHAT ARE REQUIREMENTS?

Put very simply, requirements help us capture the information intended to help solve a problem or achieve an objective. Normally – but not always – this involves defining something the customer wants. They help us make design choices, by which we mean create a solution. Naturally, this means much of the process of capturing requirements involves understanding what our customers want: things they know they want; wishes they can't articulate but which can be inferred from how they behave, and desires that they have no idea about... yet.

Ellen Gottesdeiner, an author on software requirements, puts it more formally:

> "Requirements are descriptions of the necessary and sufficient properties of a product that will satisfy the consumer's need."

To this end, what matters is not some pile of neatly written features, but the understanding of exactly what the user needs or wants. Working these things out is not always easy. As well as traditional forms of analysis, there are many ways of connecting with customers to gain insights; we suggest some in our Understanding Your Customer session.

Activity 1: Interviewing and learning

This activity is taken from the Nine Boxes Technique in Solution Selling. It helps practise ways of uncovering more about a project.

It requires a minimum of 3 people. The game works best when those playing do not work together normally. The activity should take about 30 minutes.

Roles:

The **interviewer** does not know in advance what the subject will be about. He or she needs to follow the nine boxes question format laid out overleaf.

The **interviewee** decides on a subject – a problem or project they have or are currently working on.

The **observer** will listen and watch the interviews and intervene if the interviewer breaks the rules. Afterwards the observer provides feedback on what was observed.

How to play:

INTERVIEWEE			
	What is the problem?	Who is impacted?	Visualise the solution
OPEN "Tell me about..." "And then...?" => STORIES	1	4	7
CONTROL "How many...?" "How much...?" "Where...?" => FACTS	2	5	8
CONFIRM "If I understand correctly <rephrase>. Did I understand correctly? => NO - go back to open question => YES - choose next row	3	6	9
INTERVIEWER			

The interviewer must start with questions in box 1, gradually working through to box 9. As the interviewer reaches the clarification questions and receives a 'No' he can return to an open question and work through it again.

Observers must be strict and correct any interviewer who is NOT using the correct question form or order. When making observations later, observers must also ONLY begin their comments with the phrases 'I heard' and 'I saw' – NOT with 'I thought', or 'I felt'.

Commentary:

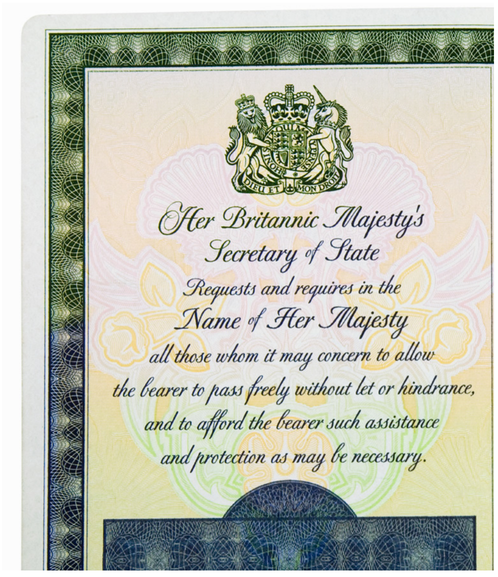
While it may feel slightly artificial, sticking to this format allows us to uncover far more information than is usual. Normally, requirements begin as if they are in the last box – confirming a pre-existing solution. There is often value in understanding the problem more deeply by going through the impact and importance of it. Sometimes you reach an entirely different solution, at other times it is the same solution, but now both interviewer and interviewee share a much deeper understanding.

2 HOW DO WE THINK ABOUT REQUIREMENTS TODAY?

Semantic pedantry can be very irritating – like insisting tomatoes are a fruit, not a vegetable, or that the singular of dice is di (both distinctly dubious arguments in any case).

In the case of requirements, however, there is a genuine reason to look closely at the word. The normal meaning of requirement colours our attitude towards the artefacts we call 'requirements' in projects.

Every British passport has certain words printed on the inside cover: Her Britannic Majesty's Secretary of State requests and requires in the name of Her Majesty… The 'requires' is put in for a reason. We're asking you nicely, the passport says, but if that's not enough then we jolly well insist. When a job advert tells you that fluent French is 'required', it doesn't mean that it's considered a bonus if you can say 'Mercy Bucket' to the visiting Parisian client. If the doctor tells you that you're not managing to get your daily iron requirement in your diet, then he doesn't mean that there's room for negotiation on the subject – he expects you to take the extra iron tablets.



Her Britannic Majesty's Secretary of State Requests and requires in the Name of Her Majesty all those whom it may concern to allow the bearer to pass freely without let or hindrance, and to afford the bearer such assistance and protection as may be necessary.

Figure 4. The inside cover of a British passport

And our requirements evoke the same understanding. They are 'necessary', 'mandatory', 'set in stone' – required, for goodness' sake!

Perhaps you congratulate yourself on being far too sophisticated to allow a literal definition like that to impact on how you approach requirements! Just consider for a moment:

- How much time goes in to gathering the information to form or flesh out requirements?
- How many people are employed in the process?
- How long do requirements last – are they kept for purposes of governance or traceability?
- Are requirements signed-off by stakeholders (and/or customers) before development begins?
- Will final delivery be judged against how well it fulfils the requirements?

Project success or failure is often judged by how it performed against original requirements. The influential Standish Chaos Report stated that challenged projects were completed with an average of only 61% of the originally-specified features and functions, for large companies the average was only 42%. This is a direct measurement of success or failure based on requirements.

This tells us something very important.

We treat requirements as contracts to be fulfilled.

Figure 5. Please sign here... here... and here

Sometimes this is explicit – as in the case of the projects viewed by the Standish Chaos Report. You may have a customer angrily insisting 'but I specified an extra widget – why isn't it there?' But more usually and insidiously, it is implicit. Your organisation may be Agile through and through, you may be certain that you accept changes to requirements whenever they are necessary – but you may also have a project backlog filled with 200 requirements that have been diligently gathered through conversations with customers and users and all our feedback is focused around how well we are fulfilling those requirements or not.

Activity 2: Rewriting history

This activity can be carried out on your own with occasional help from colleagues over a week or so.

Begin by selecting an old project – preferably one you know well enough to be able to access past information easily. Try to find the original requirements and select a random handful of these – perhaps 10.

Now, find the corresponding features at the end of the project. Did the features get descoped? Changed? Written exactly as they were originally described?

Speak to the business analyst and, if possible, the business sponsor on the project. Don't prompt them with your results, simply ask if they can remember any major changes to these particular features.

Now consider a new project. Once again select a handful of requirements. Ask the product owner or sponsor what would happen to the project if the selected features were to be descoped or changed dramatically.

Get together with your team and briefly present your findings.

Commentary:

In most cases, we find that people 'forget' the original requirements. That's fine – as work has progressed and they have learned more, ideas have changed. They're happy with the end product, so the wrong-turnings and original plans are consigned to history.

The point, however, is that most people don't think about this as a project is starting. Instead of constantly holding in mind the fact that everything will change and that only a tiny proportion of ideas will end in the same way that they began, they believe that this time all features are necessary and exist in the correct form. They find it difficult, even challenging, to have it pointed out that this is not the case.

3 CONTRACTS SEPARATE DESIGN FROM DELIVERY

The first, and most dangerous, consequence is that we separate design and delivery. The more praise-worthy work that has gone into understanding, assessing and capturing customer needs, the more reluctant we are to abandon our hard-won insights. This is in direct conflict with the concept that we discover more about the product and our customer as we work. Discovery is gradually squeezed out by a reliance on up-front design – all the harder to spot because of the work that has gone into making our requirements gathering and documentation process 'Agile'.

We are really trying to optimise for delivery, and yet by mistake, we become focused on requirements instead. We devote time to the way we write or craft our user stories; we have whole teams focused on decomposing work into small chunks (but keep them divorced from delivery), and we even go back and change our requirements in order to track what has happened. These are all signs of over-investment in the requirements process. See Figure 6 overleaf as an example of a requirements process diagram.

Requirements are a form of planning – an up-front design of what our product must do. Although we know that our understanding is going to change, because of the way we measure ourselves against this start point, requirements easily become a focus of discontent.

> "Of course we went over budget – you got the requirements wrong."
>
> "Sales are terrible because the analysts didn't correctly identify the right requirements."

Since no-one wants to incur this blame or fail on measures like those established by the Standish Chaos Report, we spend more time on trying to get our requirements right. And thereby make the problem worse.

It also means that since getting requirements right is so crucial, we stop taking risks. Don't forget the essential point we made earlier – there are unknown requirements; things customers can't say they want but we may infer, and things they can't yet imagine.

If you try to come up with these unknowns, you will be wrong a lot of the time. If the idea of being wrong is dangerous because you are seen as wasting development time or in some way not a good analyst, then you are more likely to stick to a formula where you can't be blamed. How about just asking a customer to list what they know they want?

If the customer says he wants more memory for songs on his personal music player, then you build it. It's not your fault the stupid customer goes out and buys an iPod Shuffle which has less memory.

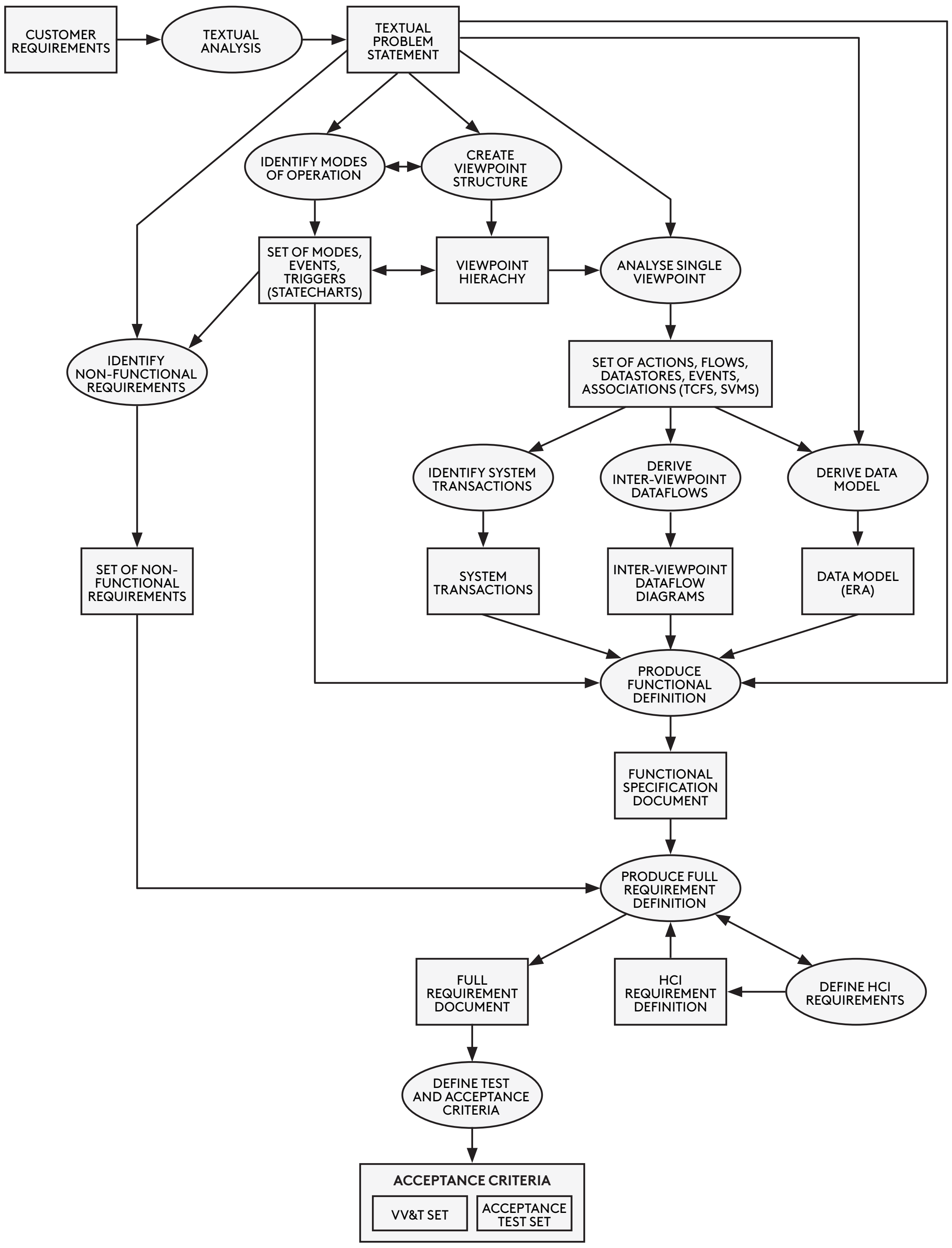

CUSTOMER REQUIREMENTS
TEXTUAL ANALYSIS
TEXTUAL PROBLEM STATEMENT
IDENTIFY MODES OF OPERATION
CREATE VIEWPOINT STRUCTURE
SET OF MODES, EVENTS, TRIGGERS (STATECHARTS)
VIEWPOINT HIERACHY
ANALYSE SINGLE VIEWPOINT
IDENTIFY NON-FUNCTIONAL REQUIREMENTS
SET OF ACTIONS, FLOWS, DATASTORES, EVENTS, ASSOCIATIONS (TCFS, SVMS)
IDENTIFY SYSTEM TRANSACTIONS
DERIVE INTER-VIEWPOINT DATAFLOWS
DERIVE DATA MODEL
SET OF NON-FUNCTIONAL REQUIREMENTS
SYSTEM TRANSACTIONS
INTER-VIEWPOINT DATAFLOW DIAGRAMS
DATA MODEL (ERA)
PRODUCE FUNCTIONAL DEFINITION
FUNCTIONAL SPECIFICATION DOCUMENT
PRODUCE FULL REQUIREMENT DEFINITION
FULL REQUIREMENT DOCUMENT
HCI REQUIREMENT DEFINITION
DEFINE HCI REQUIREMENTS
DEFINE TEST AND ACCEPTANCE CRITERIA
ACCEPTANCE CRITERIA
VV&T SET
ACCEPTANCE TEST SET

Figure 6. An example of a requirements process

The second problem is that we end up with too many requirements and features – many of them valueless. Especially if you say this is a customer's only chance to specify what they want, most customers will throw together an extensive shopping list of anything they might need, want, or have heard about at a dinner last week. At a superficial level this is 'talking to the customer', but it's not the same as 'understanding your customer', in which you search out an underlying need to which you might be able to create an unexpected, desirable solution.

As Gojko Adzic put it, 'This often results in software that does what the customer asked for but does not deliver what they really wanted or is functionally incomplete.' It is also a surefire way to avoid that favourite cliché of the tech industry "game-changing innovation". Instead you'll be limping after more daring competitors.

Activity 3: Origami discovery

This is an exercise to do with a group – it can be a fun start to a weekly team meeting or a planning meeting to prove a point. It should take only 15 minutes.

You will need 6 people, divided into 3 pairs.

Print out three copies of the origami instructions (you can download these from the Additional Resources pages of valueflowquality.com).

Each pair has a set of instructions (laid face down) and several sheets of paper for folding.

- **Group 1: sits side-by-side.** Only one person will fold, while the other advises, but both can consult the instruction sheet as much as needed.
- **Group 2: sits face-to-face.** The folder must not see the instruction sheet, but the instructor can describe what needs doing, watch the folder and provide feedback on progress.
- **Group 3: sits back-to-back.** The instructor and the folder cannot see each other. The instructor must describe the actions required, while the folder can ask questions in return.

As soon as the origami design is completed, the group should stand up. After 15 minutes, the exercise ends (even if not all the groups have finished).

Commentary:

A perfect set of instructions does not necessarily translate into perfect construction. Collaboration, shared understanding and learning through trying are far more important and helpful than making the instructions themselves clearer or better.

It may be worth pointing out to the group that the actual requirements process might be more like the instructions arriving by email or being described over the phone – even harder than any of the situations simulated in this activity.

4 HOW SHOULD WE THINK ABOUT REQUIREMENTS?

Requirements, we said, capture pieces of information intended to help us solve problems or achieve objectives. There are some difficulties inherent in this that we need to remember:

1. Not all information is equal
2. Information has a lifecycle
3. Information can overwhelm action

4.1. Not all information is equal

People don't know. People omit details. People forget. People confuse facts. People exaggerate (or lie). People change their minds.

All of those failures of information quality have a major impact on the quality of our requirements, and thus on our eventual product. Let's take the most extreme – a lie.

How often have you seen some numbers gently massaged in order to get approval for a project? You may have even done it yourself. Boosting a project over the minimum rate of return a company is willing to accept before starting a project (a hurdle rate) couldn't feel very serious (many organisations do it) – and yet from that initial piece of information, decisions will be made that have serious impacts on the project.

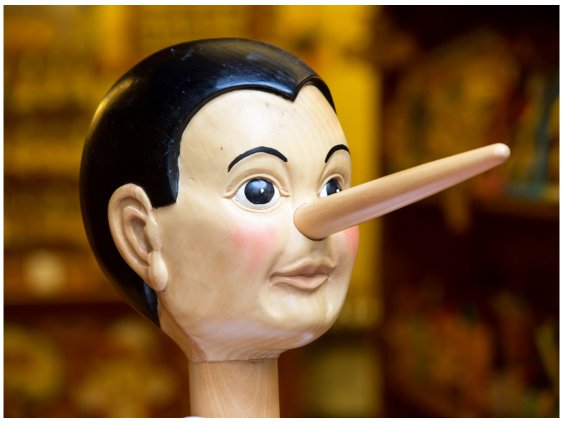


Figure 7. Lies are easier to spot in some people (image copyright © nevenm)

Question: How much will your application make? (P.S. we are only funding projects making £20 million+)

Answer: £25 million. (Hmm I'm planning to charge £1 per user, so I better have at least 20 million users. We got 15 million on the last product – and that was rubbish – so I'm sure we'll be able to get 25 million.)

That estimate will form an important part of the non-functional requirements. In order to manage that number of users, the company may need to start buying servers, hire space to put them, increase bandwidth, scale the database…

But how reliable is the requirement to build for 25 million users?

Once a piece of information enters the plan, it often goes unchallenged. We need to accustom ourselves to interrogating information and testing to try and discover if it is correct or not.

CASE STUDY: Mission to Mars

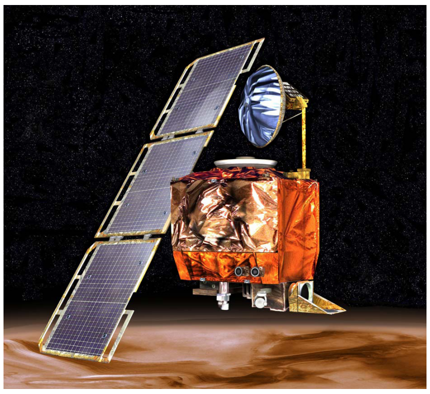


Figure 8. Artist's rendering of the spacecraft Mars Climate Orbiter

The information we don't have can sometimes be more crucial than the information we do have. In November 1999, the $12.5m Mars Climate Orbiter (MCO) entered the Mars atmosphere at too low an altitude and disintegrated. It would be another 7 years and $270m before the next spacecraft would be able fulfil the MCO's mission.

Losing a spacecraft is upsetting enough – discovering that you've lost it because of a very embarrassing failure in requirement writing is even worse.

There were two engineering teams working on the software systems for the spacecraft. NASA's team created the flight system software, while the global aerospace contractor Lockheed Martin wrote the software on the ground, which controlled the spacecraft. Naturally there had been enormous amounts of testing done to check that the two systems were compatible. But the tests missed something – something so obvious that no-one even thought to bring it up.

The NASA team had used the metric system regarding thrust instructions, measured in newtons (N), while the software that generated those instructions used the imperial measure pound-force (lbf). That meant all the careful calculations, tests and final instructions were based on a false assumption. The Mishap Investigation report concluded, 'subsequent processing of the data from the AMD file by the navigation software algorithm therefore, underestimated the effect on the spacecraft trajectory by a factor of 4.45, which is the required conversion factor from force in pounds to newtons. An erroneous trajectory was computed using this incorrect data.'

With commendable restraint, NASA called the mistake 'the metric mix-up'. They put in place actions to 'avoid' it in the future. Of course that might include a standards document insisting on the engineering measurements to be used by all teams, or it might be as simple as having the teams review one another's working.

4.2. Information has a lifecycle

Raw materials last for differing amounts of time. If you are a sculptor you can be relatively certain that the marble block standing in your yard is not going to crumble to dust while you decide whether you want to sculpt Michelangelo's David or the famous Kiss statue by Rodin. If you have a net of live mussels then you had better decide what to cook quickly. Information as a raw material sits somewhere in between marble and mussels. It has a lifecycle and it has different values and costs associated with it, depending upon where you are in your development cycle and when you receive it.

In some examples this is very obvious, even binary. Knowing that a two-year-old horse has been recording some fantastic runs is highly valuable before the Grand National horse race. After the outsider has romped to victory, your tip is valueless.

In other cases, information's value is on a sliding scale:

VERY VALUABLE: You are handed a Top Secret memo with your competitor's plans for a year's time. They have a killer new technology they hope to launch then. You might decide to develop your own version and launch it earlier.

VALUABLE: You are six months into a normal development cycle and you get the memo saying your competitor is ready to launch something next month. You might try to rush your own production or you might delay it, waiting to see if you can copy whatever your competitor's features are and launch an even better product.

USEFUL: You hear on the grapevine your competitor is due to launch next week. You can send out a 'spoiler' press release, saying how great your own product is.

USELESS: You receive the press release about your competitor's launch. You can all go down the pub to complain about how rubbish their UX is.

In other words, information has different values depending on when you receive it. It also has different costs associated with acquiring it. This is true for all information – not just the results of industrial espionage.

Reliable information at the very beginning of the development process is valuable, but it might cost a lot to acquire. That might mean costs in doing market research or taking senior business people's time away from their day-to-day jobs. It also means that the information you have gathered with such difficulty and cost doesn't stay good forever.

Information has a half-life and its validity decays with time. The trick (and it really is a difficult trick) lies in balancing the validity, cost and value of information throughout your development process. The painful paradox is that the more time you spend acquiring information, the more likely it is to lose its value through being out of date.

Activity 4: Acting out

This game works with between 4 and 8 people and makes a good warm up for a team meeting. It takes approximately 5 minutes.

Select someone to be the designer. The designer should write an action or description down: being a magician at a children's party; throwing confetti at a wedding; changing beer barrels in a pub cellar.

Everyone else playing needs to turn their backs or leave the room.

The designer calls the first person in and hands them the card. The first person calls in a second person and acts out what's on the card in 30 seconds. No-one can speak or make any sound effects.

The second person calls in the third person and acts out what they have seen, or what they think the action represents in 30 seconds.

This continues until the last person has been called in. The last person then tells the designer what they think was written on the card.

Commentary:

Rather like the Chinese whispers game played as children, the point of this activity is that information decays as it passes from person to person or from one medium to another. We may think that we are more sophisticated than a party game, but in actual fact miscommunications and misunderstandings are a common cause of errors, rework and significant frustration.

4.3. Information overwhelms action

The amount of information in the world is increasing, as are the number of ways in which we can access it. This shouldn't be any surprise to those who work in IT – just in case you had forgotten the initials stand for Information Technology. Yet, quantity does not mean that we can do anything with all this information.

We are fast approaching the age of the zettabyte of information. The raw material of information that we put into our requirements is growing exponentially, as are the systems that we are building to use it – all of which means we are flooded with still more information.

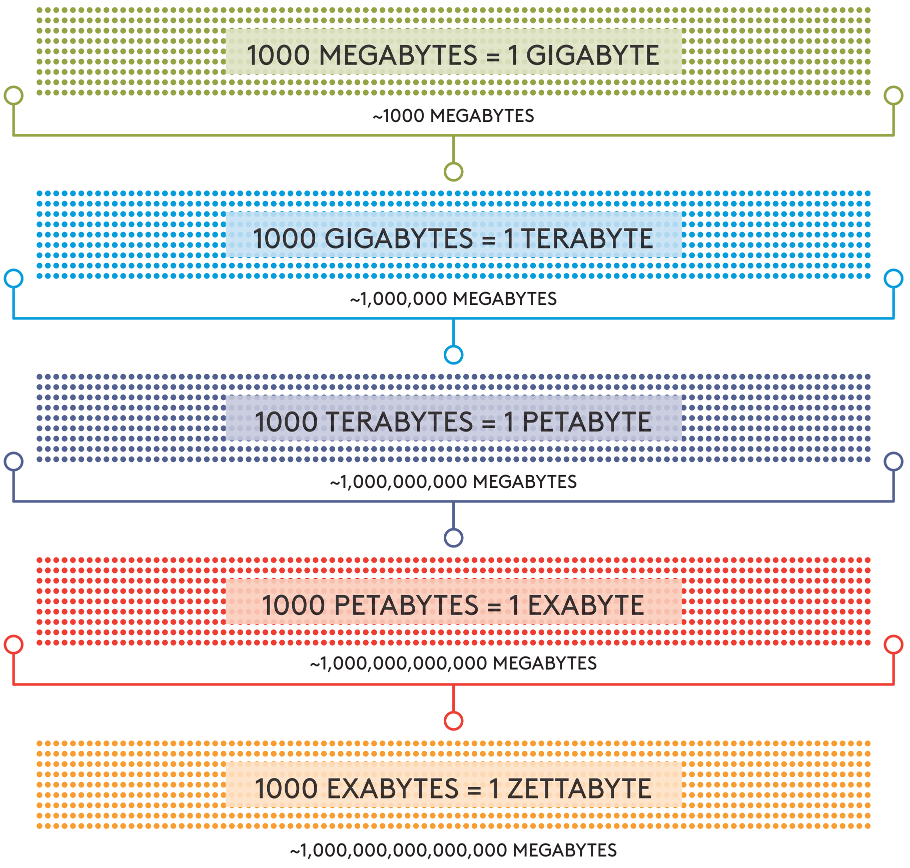

1000 MEGABYTES = 1 GIGABYTE
~1000 MEGABYTES
1000 GIGABYTES = 1 TERABYTE
~1,000,000 MEGABYTES
1000 TERABYTES = 1 PETABYTE
~1,000,000,000 MEGABYTES
1000 PETABYTES = 1 EXABYTE
~1,000,000,000,000 MEGABYTES
1000 EXABYTES = 1 ZETTABYTE
~1,000,000,000,000,000 MEGABYTES

Figure 9. The size of a zettabyte

Consider an encyclopedia. 20 years ago, the Encyclopedia Britannica encompassed 29 volumes, comprising roughly 40 million words and 24,000 images. As this session is written, Wikipedia has 4,076,832 content pages and 791,996 uploaded files. By the time this session is printed both numbers will have risen.

There are new types of data available. Direct mail companies used relatively sophisticated ways of keeping track of orders, checking if grouping items together in a catalogue boosted sales, for example. But their work pales beside Google Analytics and our ability to measure how long people spend on each web page, or see each step in a transaction.

We can trace people moving through airports or shops, we can even use eye-tracking software to see what parts of documents or webpages are read, skimmed, or skipped by the user. In theory, we could watch you reading this session (with your permission, of course – we don't actually have a secret spy camera in your home) and observe your reactions to each paragraph.

All this data tends to result in more requirements. As our systems need to hold and manage so much more information, they get larger and the things we want them to do also grow exponentially. The techniques that used to work for us in managing a few hundred requirements, break down when faced with hundreds of thousands, perhaps millions of requirements.

Once it was possible to create a complete, exhaustive specification document of a few dozen pages – now the very idea of completely synthesising all the information that might be useful is laughable.

Activity 5: Spot the difference

You can do this activity at any time, either on your own or working together with a couple of colleagues. Watch the following video for a cyclist awareness advert from the UK. See how many deliberate changes you can spot.

http://www.youtube.com/watch?v=ubNF9QNEQLA

Commentary:

How well did you do? Even when you know in advance that there are deliberate changes being made, ones which you are concentrating on spotting, it is almost impossible to see them all. An overload of information overwhelms us – even when we think we're prepared.

4.4. Summary

The difficulties of information leave us with a problem. Requirements are valuable pieces of information with a fairly short half-life. We need to move quickly in order to keep the value of the information. We also need to know that we have captured the most valuable pieces of information and not the mistakes, lies or environmental 'noise'.

Information doesn't tell us what to do. Gathering information takes time, costs money and can stop us getting on with producing something (whether that's writing working software or researching a new session for VFQ). The sheer quantity of data available can leave us paralysed before we begin. From the mass of information out there, how can we possibly hope to extract the bits we need?

5 THE AGILE REQUIREMENT

Agile attempted to tackle some of these problems head on. It enshrined the idea that we should 'welcome changing requirements, even late in development'. Enabling that meant delivering frequently, working daily with business people, conveying information through face-to-face conversations and using the working software itself to help show what's wanted and what is not rather than 'comprehensive documentation'.

All of which is brilliant. It is also not necessarily simple. In this section we are going to examine the most commonly employed tools in Agile for discovering, capturing, storing and using valuable information. We are also going to briefly explore some of the ways in which the tools – used with the best of intentions – can get in the way and exacerbate the very problems they were intended to solve.

5.1. User stories

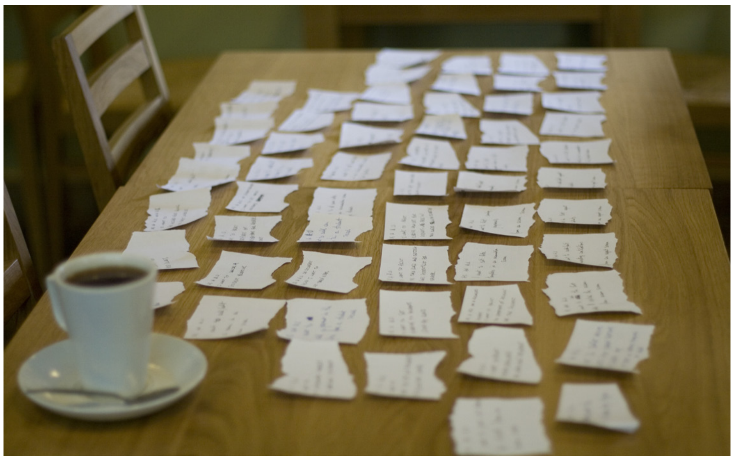


Figure 10. User stories laid out on a table (image by Jacopo Romei)

The user story has become the workhorse of an Agile project. Barely a software department today doesn't use a large number of cards pinned to walls or Post-it notes scattered across windows in varying states of disorder.

A useful reference for user stories is the XP practice known as: Card, Conversation, Confirmation. It has rather fallen out of fashion as a description of the process, which is a shame because it is helpful in pointing out the essence of how user stories should work. The practice does not suggest a single sequential round, but rather a cycle, in which the differing elements can run in parallel as the development progresses.

The card is simply a placeholder – a visual attention-grabber with some basic information that serves as a prompt for and reminder of the more important conversation. It's been called a 'promise for a conversation'.

The card alone is insufficient. The most crucial part of the process is the middle element – the conversation in which details will be fleshed out, performance considered, and numerous other connections made to how a task helps solve a problem or captures a benefit.

We'll come on to the confirmation part below in Section 5.5.

Figure 11. Post-it reminder (image by Paul Downey)

When user stories follow the card-conversation-confirmation stages, they become a low cost way to capture information and store it. They can help reinforce the concept of requirements as pieces of information and not contracts. Finally, they are easy to use.

Despite this, user stories have problems. They don't easily cover many things, including performance parameters, interactions between systems or entities and a host of other non-functional requirements (which are crucial but must often be recorded separately). The user story can never be seen as a substitute for the conversation, which can make it difficult as a tool in large teams. And lastly, their disposability means they provide little traceability – this might seem a very un-Agile need, but there are certain safety or security systems in which tracking changes to requirement input as well as to code can be important.

Writing and using user stories: the card

A user story can be anything. It is intended as a reminder allowing information to be recalled later.

In a small team all that's necessary might be a singe word: Testimonials!

But in bigger teams, or when it might be some time before we get to the card, some extra information might be useful.

It should probably say who is requesting the story: Jeff B.

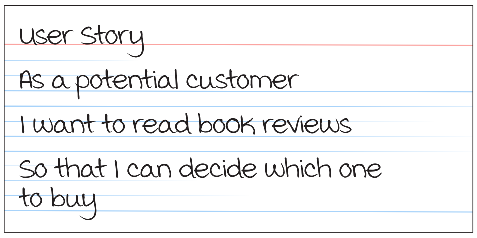
It might also add why the idea is useful or valuable: Improve conversion rates.

The value is important because it can help us to prioritise stories. The bigger the team or project the more important this is. It might be six months before your particular card gets selected, so you need to put enough information onto the card, so that someone can either talk to you about it and you can remember or it's self-evident to those reading it. Sometimes this connects to the confirmation part. In our example, we might add: Other sites have seen 30% increase in PayPal click-through. We want to match this!

The card might have any number of other useful bits of information attached to it – which type of customer it's for, a quantified value, a rough estimate… but the key is not to get too caught up in these. Rather than perfecting user stories or requirements we need to ask ourselves how to be sure they are what Scott Ambler calls 'just barely good enough'. What is the least amount of information that we need in order to get started?

Despite the complete freedom theoretically offered by user stories, in practice most teams follow a defined template in writing them:

As a … new user
I want to … register
So that … I can download the game


User Story
As a potential customer
I want to read book reviews
So that I can decide which one to buy
Figure 12. An example of a user story card

This way of writing user stories describes an intent – something which somebody wishes to do or experience. This means they do try to address value – the 'so that' part of the template.

What they shouldn't include is how the system will behave. Putting in the solution, whether technical or design, removes the developer's contribution. Sometimes, with very well-defined problems or understood solutions, that might be an acceptable shorthand, but it risks missing out on innovation and improvement.

User stories also don't tell us whether it is possible to fulfil the desire. The idea may be technically impossible. The user story can't tell you this up front.

A user story is kept deliberately brief – it is written as a few lines on a piece of card because it is designed to remind us that the story is simply a placeholder for a larger conversation. Some have gone further and suggested we reduce user stories to a single word – a device to ensure people have to talk to one another.

A lot of worry about user stories – are they too big, too detailed, too vague – is unnecessary. These are problems that can be sorted out in the conversation. If you are spending hours crafting a user story then it's a sign something's wrong. Yet, if the issue is really bothering you, perhaps because you know there will be a long gap between writing the story and working on it, then you may find the following model helpful.

Bill Wake coined the mnemonic INVEST to describe the components of a good user story.

I – Independent: The user story should be self-contained without dependencies that link it to other stories. We should be able to add the piece of functionality onto the product separately and in any order and still have a tested, usable product afterwards. This is not easy and we often do it in order to keep our stories small (see the S below) and to capture the most valuable part first.

I	-	Independent
N	-	Negotiable
V	-	Valuable
E	-	Estimable
S	-	Small
T	-	Testable

As a user, I want to pay using Mastercard, Visa and Paypal.

These are actually two tasks bound up in one, and they could be split out.

As a user, I want to pay using my Paypal account.
As a user, I would like other options to pay with a debit card.

N – Negotiable: User Stories can be changed and rewritten as we learn more. Details are 'co-created by the customer and the programmer during development' as Wake puts it. To take our example above about the download speed versus engagement, as soon as the developer realised the feature was proving tricky, we'd expect him to call up the person who'd requested the feature to discuss the problems and consider new or alternative solutions. By keeping flexibility on both the many different requirements and solution options, we can select an approach which has the best trade-off.

V – Valuable: A story must deliver value to an end user. People sometimes get hung up on this, worrying that each story should offer end-to-end functionality in a way that seems to make every story big once again. The point is to think about how a story or feature contributes to end users in some way. Will making our administration faster free up time to help users? Yes? OK, then there's value in this story.

E – Estimable: Wake is clear that there is no need for an exact estimate. Rather the customer needs enough information to be able to prioritise the story. That means that as the developer discovers more about the story in the negotiation phase, he will understand enough to assign a rough estimate. Also it implies that those writing or thinking about user stories should have enough domain knowledge to be able to understand what's involved.

In many cases we may not know how to estimate a story because the technical solution is very uncertain or the idea itself is very poorly defined. A spike might help the team discover more, at the end of which they could define the potential solution more exactly and estimate the story. The desired result, however, is the learning, rather than specific accuracy of estimation.

S – Small (or Sized appropriately): User stories should not be so big that it is impossible to plan or prioritise them. If a story seems likely to last for more than a single iteration then it probably needs to be broken down further. See the next section, Decomposition, for more detail.

T – Testable: The story must contain enough information to make test development possible. How will we know when it is done? How will we know if the feature does what the customer wants it to? These need to be considered before we start writing code although they can be added to as we learn more in the process of development. We go into this in detail in our Confirmation section and when discussing acceptance criteria. These are useful not only as a check that we have done something correctly, but because they facilitate challenge and argument – especially what level of accuracy will be considered success.

A quick warning...

While the INVEST mnemonic is useful, it is often at its most useful when considered as part of the Conversation element of user stories (discussed in the Elaboration section), rather than the card. Often the development team has expert knowledge that needs to be brought to bear to ensure we are providing what the customer really wants. Too much work upfront on checking the story perfectly fits each check makes it harder to view the story as disposable and changeable.

Activity 6: User stories and the INVEST model

The following activity can be done with a group of 2-10 people. It should take an hour.

Select a project which has either just started or is about to start - preferably a project that does not yet have user stories. It is fine if the people you are running this exercise with don't work on that project, it's just good if they don't already have a clear idea of the existing user stories.

Take a section of requirements. Ask the team to write user stories. Deliberately give them no more guidance than 'write them' on how to write the stories. You want to know how they currently go about writing them.

After about 30 minutes call a halt and put up those they have written on a board. Now using the following list of questions, ask them to reflect on what they could improve or change about the stories they have written and the process of writing them.

I – Independent: Is the story independent or does it rely on something else being completed? You might need to change your design to be more hierarchical and less of a chain of dependencies.

N – Negotiable: Is there room for any additional creativity from the team? Has someone already set down how the finished solution might look, or how the user will interact with it? If so, you may be tying things down too much and missing out on alternative ways to capture the value.

V – Valuable: Is this story valuable? Will it contribute towards our end user or customer in some way? Is there enough information here for the customer to be able to prioritise it? More importantly, check that you have captured the heart of the value and that you are not gold-plating. Have you proposed a solution that is more than the end user has strictly asked for? This is especially relevant if you have proposed a technology solution that would be helpful for the future. Ask yourself if you are looking ahead rather than concentrating on what's needed now. Anything extra – even if it feels like a good idea – will take time to implement, document and test, and we don't know if it's necessary.

E – Estimable: If you asked everyone in the room to provide a guess for how long it would take, would they feel confident that they could do so? If people are reluctant then ask what pieces of information are missing that would enable them to make an estimate. Make it very clear that you're not demanding accuracy, just whether people feel they could put some kind of estimate down. If not then the story is either too large or missing something important.

S – Small / Sized appropriately: Too small – would the story add some useful functionality by itself or is it simply a task that can be completed by a specialist within your team? Do you think you could do it in less than a day?

Too big – if you were estimating this, do you think it could be done within 2 weeks? If not, then you should consider splitting it. Think about splitting along value lines. It's better to have something small completely finished than something big, half finished.

T – Testable: Will you know when you've finished this? Will you know whether you've been successful or not? Can you capture a reasonable number of the critical tests that will provide this information on the back of an index card? If you don't know then you may not understand the user story sufficiently.

Commentary:

We hope this exercise was useful, but the point is not to 'perfect' user stories, but to create useful placeholders for conversations. A lot of the questions asked above could happen while working with the user stories themselves rather than when writing them in advance of development. The INVEST model will not produce perfect requirements, any more than a lengthy specification ever did. Use it as an aid if people are struggling with the idea, but remind them that it does not remove the need for conversation, adaptation and collaboration.

Problems with user stories

Used well and appropriately, user stories cut down on unnecessary documentation, remind people of important Agile principles and support just-in-time planning and prioritisation. Unfortunately, it is easy to misuse user stories – primarily by forgetting what they are meant to be and instead treating them as if they were a specification by another name.

User stories are not requirements. They are visual reminders intended to capture minimal information which might help solve a problem (the requirement). They point to possible value. That's pretty long-winded, so no wonder people conflate the ideas: user stories represent requirements, ergo user stories are requirements.

But like most simplification, it leads to a problem. Dan North wrote an insightful blog post 'The Perils of Estimation' where he comments on how the process of breaking an idea down in order to say how long it will take means getting into such detail that we lose track of the original purpose. 'The business started out by defining success as solving the problem, but now we have redefined success as delivering this list of stories.'

Figure 13. User stories begin to take over the office

The ideal user story may tell us 'why, not how', but in practice this is rarely the case. Instead, our reliance on features means our user stories get filled with pre-formed solutions – as a user I need a password so that my account is secure; as a user I need an account so that payment can be taken automatically. Both of these seemingly correctly written stories contain hidden solutions.

A password is not the only, or even the best, form of security. Automatic payment may be useful for the organisation, but is it really what the customer prefers? The stories have created constraints, which may hinder or stop us uncovering solutions that might actually deliver the real value. These pre-formed solutions might also cost us a great deal to deliver (that's especially true when they constrain the system rather than just a single feature). Behind these solutions, the real requirements are hidden away at a higher level.

User stories try to tell us why the feature is there (the so-that element), but they tend to miss the broader intention. This context can be crucial for delivering value. It includes considering how long we want to take to reach our goal – something that it may be important to capture as part of the requirement because it will influence our design. If it is essential that my feature is 100% accurate from day one, for example, then I will make different design choices than if my aim is 100% accuracy but I am prepared to begin with 90% accuracy and iteratively improve.

The language of user stories is often ambiguous. We try to correct this through the conversation and confirmation elements of the practice, which clarify and elaborate on the card. But since the card exists before the conversation and might get separated from it, the language risks being misinterpreted by numerous stakeholders.

Occasionally, of course, we have no information to note down yet. But when we do happen to know more about risk, cost or value, it is worth recording our assumptions. Refusing to put any of it down because we are wedded to a simplified user story template risks defeating our purpose.

There is nothing about a user story to tell us it is right. It's very important that we bear this in mind – we can write code that passes the user story's test and still not have delivered anything of value to the project. As Tom Gilb points out with great force, when projects fail, it is not usually because they lack functionality, but because the product qualities don't deliver what stakeholders really value. Meanwhile the focus on fulfilling the (dubious) functionality has often led to uncontrolled costs. Sadly, this is not a disease to which Agile is immune.

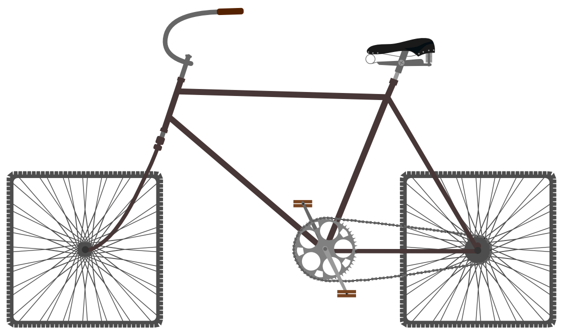


Figure 14. Bike launches with revolutionary functionality! Sales mysteriously low

We'll explain the key problem we see in Agile requirements in our next section.

5.2. Decomposition

If there's one thing that Agile teams have really grasped, it's the concept of breaking stuff down. Want an online shop? We can break that down and then deliver it through increments! We'll need a search function, ordering results, an account system and database, a gallery, a price database, user reviews, sales reporting, automatic stock ordering, delivery options, automated emails, targeted discounting offers, loyalty points…

We all understand why we'd do this. A great big huge project is just too big. The risk is that we take too long, lose sight of what's most valuable and instead launch everything in one big bang, massively increasing our risk. To power incremental and iterative delivery, as well as to improve flow through small batches, we break work down. But (there's always a but) sometimes we get it wrong. In our online shop example, all the features are in danger of obscuring what's really critical – enabling an individual customer to purchase a product. We'll look at the issues – and the possible solutions in a minute.

Using story decomposition

As we break work down, we tend to use the following rough order.

Order	Example
Business Goal: the reason for doing the work – must add value	Improve conversion rates
Minimum Marketable Feature (MMF), sometimes called Minimum Viable Product (MVP): the minimum product that we wish to launch to the customer	Redesign website home page
Epic: a valuable idea containing several inter-related and connected pieces of work	Add testimonials section with gallery and pictures
Story: a single improvement or feature that the customer will find valuable	Add text testimonial
Acceptance Criteria: how we will know if we have captured the value	Text testimonial can be read on different browsers

Obviously there are several Minimum Marketable Features (MMFs) that can go into improving conversion rates, just as several changes or additions must be prioritised to decide which element of the home page will be redesigned first.

There will be a lot of ideas as to how we can improve the home page – from adding extra features on it to improving clarity of layout. We can often prioritise between these ideas (here described as epics) before there is any need to start decomposing them into actual stories or tasks. If we don't plan to work on the testimonials, for example, there would be no point in splitting the epic down any further until we have decided that it is time to work on it.

Having selected the 'epic' we wish to work on, it can be split down into stories. Sometimes we skip steps, but the idea is that at each level, the number of possibilities to fulfil the goal above increases (rather like a pyramid).

Please note that this order does not always hold true – occasionally, for example, MMFs may be very small indeed, consisting of only a single story! If the customer will genuinely find just one small thing valuable – then we should probably release just that. In other cases, especially where we want the customer to buy something or change their normal way of interacting, for example, the MMF might be much larger and consist of a large number of epics. As always, the key word in the acronym to focus on is 'minimum'.

Splitting work

Long before we begin decomposing work into small user stories, we need to think of slicing big and complex problems that deliver the most value to our organisation. This subject is covered in detail in our Deliver Early and Often session. The main 'prisms' that let you look for and then split out different types of value are: Value, Urgency, Risk, Stakeholder, Geography and Necessity. For example, we can consider what will be most relevant to a particular user group; we might prioritise English-language users or a single country's currency before worrying about other versions. That would tell us which large ideas (epics) we need to focus on.

We can often draw out one particular strand with independent value that crosses these large ideas. A useful technique is often to think about a user's process steps – a customer journey in the system. For example, if you were thinking about an account system, you might think about a user creating, accessing, editing and deleting his account. Each step might have its own series of user stories. If you think about the things a user might want to do while accessing their account – change address, credit card details or other data – then you have a further series. Differing levels of quality or complexity can also provide separation points, allowing us to build iteratively better products.

The fact is that how you decompose the work tends to be domain or context specific. Story decomposition is intended to ensure that by the time we get to actually building something we can see the connection between the small task we're working on and the larger business goal that it supports.

The risk with decomposing work too quickly is that we lose this connection and don't even notice that we've gone off track. It's surprisingly easy for this to happen. Imagine we had a series of steps about a user accessing his account. Oh that's easy, we might say, we know how to do all the log-in and password stuff, and we would quickly write a load of user story cards. But is a login the correct solution? Does it really answer the underlying business and user need around security or is there a neater way?

Decomposition at scale

At scale, we might be talking about breaking work down across multiple teams. In this case, while similar processes of decomposition will be happening over the project as a whole, the breaking down of work might not be done by the team who will actually be doing the work itself. Instead architectural and business epics will be passed down to a system team who will then pass features and stories down to specific teams.

The realities of large structures may mean that occasionally work might be broken down by technology or by specific team function. Since that means two teams might be working to deliver the eventual user value, co-ordinating their efforts becomes even more important.

Feature injection

Feature injection, as popularised by Chris Matts in 2008 and elaborated over the last few years by others including Liz Keogh, tries to help preserve the link between business value and user stories.

The aim is that the development team should understand how each feature directly delivers to business value. To this end, they first 'hunt the value'. Only once this has been established do they 'inject the feature' which will best deliver that value. Finally, they 'spot the examples', which will feed in to what the developers actually work with, since it confirms whether the feature is delivering the value or not.

In theory, this should stop an excess of features, many of which don't contribute to the value or needs of this particular project or business aim. This, to some extent, reduces the need for a 'Confirmation' stage of acceptance criteria or specification by example, which is described later. There's nothing revolutionary about feature injection as a tool, but it can offer a good way to check that you're not breaking things down into tiny features as an end in itself.

Let's imagine a recruitment agency – the classily named, Jobs4U. They operate a 2-tier service and are keen to upgrade more users to their premium, paid-for accounts. They have discovered that once they have a user's email details registered, they can send marketing emails and offers to achieve a much higher conversion rate than simply through advertising the service on the site alone. This insight leads to a clear statement of business value.

Hunt the business value:

Jobs4U want to capture more email addresses so that we can sell more 'premium access accounts'.

Inject the feature:

As a user, I will provide my email address, if Jobs4U offer to prompt me so that I don't miss the application date on jobs I am interested in.

Spot the examples:

Given I have clicked on a job advert, when I have been on the page for 30 seconds, then I will be prompted to enter my email address to remind me about application dates.

Given I have entered my email address into the email address field, when I press submit, then the result will be an entry into the database.

Given I have a registered email address, when I click on a job advert to show my interest, then the result will be an email reminder from Jobs4U the day before the application closing date.

The problems with decomposition

There's something slightly addictive about breaking down ideas into stories. After all, how else can the team provide relatively accurate estimates of how long and how expensive a project might be? And no matter how much we discuss the benefits of no estimates, many organisations still need to answer the question, 'if we want to build this, then how long will it take and how much will it cost?'. If viewed as a discrete task in itself, decomposition means that one team of analysts and customers can break everything down, before handing over lots of fine-grained stories to the development team to start flowing through.

Of course, it leads to all the problems we've already discussed – it takes too long, separates design from delivery and drives out discovery and innovation. It sets up the dangerous assumption that this lengthy list is what the project will actually deliver (thus setting the team up for failure and disappointing expectations). But it also obscures the real value.

This can seem crazy, because the organisations that do this, also often have people assigning value estimates and prioritising the backlog on a weekly basis. Nonetheless, this vast accretion of stories buries the essentials – what we'll call the critical features. Sometimes this is because the critical features exist at a higher level, and are drowned in the minutiae or thousands of little tasks. Sometimes, it is because in our urgency to capture everything that might contribute to the business value, we lose sight of what matters most to it. Using tools such as feature injection will not avoid this problem, because we can still become obsessed with all the possible features to provide value.

The result is unattractively but colloquially known as 'the vomit': a huge splurge of everything we can possibly think of, broken down to a detailed level. We then have to manage this morass of information, spending time valuing, estimating and tracking the results (a costly overhead in terms of our time). So how do we avoid this?

Activity 7: Breaking down

This game is best played with 4-5 people, although you can have a number of teams competing against one another.

You will need 2 piles of different coloured cards or Post-it notes.

Part 1

Begin by standing in a row. The first person should write down what he wants "pizza", "dinner", "cocktail", "Harley-Davidson motorbike" – whatever it happens to be.

He passes this card to the next person along who needs to break this down a level to the key things required to make this. To take our pizza example, he might write (one thing on each card): pizza base; tomato sauce; toppings; pizza oven.

He passes his cards along to the next person who begins breaking each one down again before passing them along. For example, pizza base might be: flour, eggs, yeast, olive oil. Sauce: tomatoes; oregano; onions; garlic. Toppings: spinach; egg; cheese. Oven: bricks; clay, firewood.

Once these cannot be broken down any further, the last person(s) needs to begin fulfilling the requirements. He does this by drawing the item required on a different colour card and then passing this back along the line. A sketch of a garlic clove would be attached to the card 'garlic' and passed back. The next person waits until he can collate everything needed for 'sauce' before passing this back. This person waits until he collates everything required for 'pizza' and then passes it back to the original requester.

If you are doing this as a series of teams then the fastest team to collate everything back at the start point wins.

Commentary:

The activity shows several things. Although the process of decomposition is useful in some ways – it might provide us with a neat shopping and equipment list for our pizza, for example – it is pretty slow and you end up with an awful lot of paper.

It's easy to miss things, as well. People further down the line make design decisions for those up the line. Did our requester want a fiorentina pizza? If not, he might be pretty disappointed with what turns up. We went to quite a lot of effort and expense to make our own pizza dough and build a wood-fired pizza oven. Would he have been just as happy with a pre-made base cooked in the electric oven? If so then we've wasted a lot of resources.

Part 2

You can play the game a second time. This time, instead of standing in a line, huddle around a table. Talk as you work. As soon as one requirement is written it can be passed on to be broken down further. People can question the original requester – did you want a particular flavour? Are you vegetarian?

At the end, everyone can check nothing's missed out. What about knives to chop up the toppings? What about salt?

Commentary:

You should play the game much faster the second time. The collaboration not only speeds things up, but it also provides a better solution for the original requester. Some teams go even further and start questioning the original request. Are you sure you want a pizza made? How about a takeaway? How about this apple?

It helps show how collaboration from the team can help improve even the original 'requirement-solution' by understanding more about what prompted it, allowing the team to deliver faster – and thus win the prize!

5.3. Identifying and prioritising critical requirements

Jason Fried, the founder of 37 Signals, talks about identifying the most important elements of a product, stripping out all the potential functionality until you uncover the core, the thing without which no product or functionality can exist. He calls it 'epicentre design'. He wrote a blog about creating a calendar for users, and described some of the many add-ons the team considered in design. The heart of the feature was deceptively simple, however – a calendar requires the ability to create events. Without that very simple functionality, nothing else has any meaning.

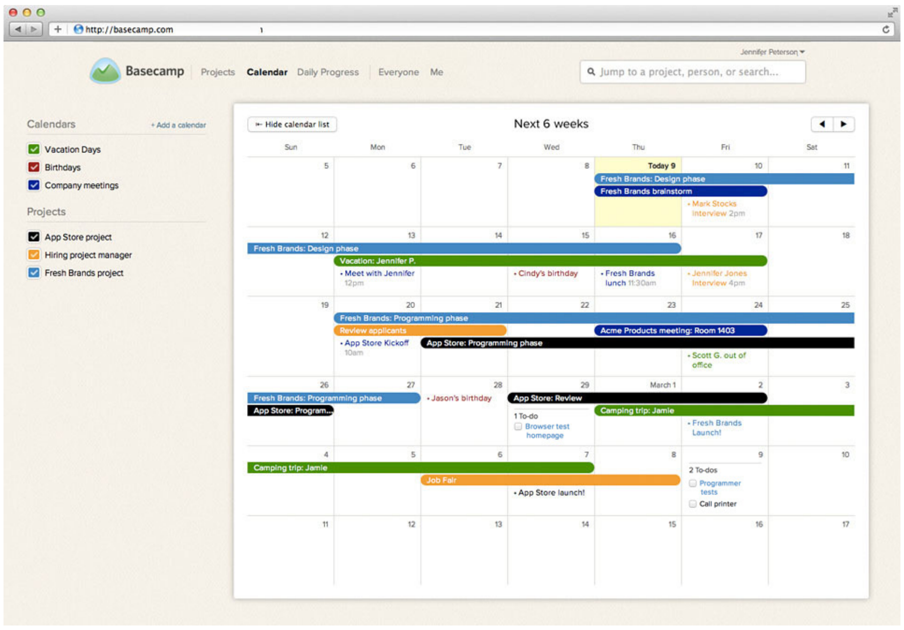


Figure 15. An example of the Basecamp calendar

In the example we gave above of building an online shop, we could strip away almost all of the features that we listed. The epicentre design might be to show the product alongside an email address or phone number. Orders could be handled manually. The critical requirement does not include the ability to have a basket, an account or even a payment system. In fact if nobody wants to look at or buy our product, then spending any money on developing an online sales system is crazy.

This may seem extreme, but in fact many requirements are piled onto a project long before we know if the critical requirements will produce any value at all. Tom Gilb calls it an over-reliance on features – creating numerous pre-formed solutions to problems we have not yet fully defined. Our critical requirements expose our fundamental business hypotheses and by testing these, we give ourselves permission to move on to the next increment.

A focus on high-level or critical requirements helps explain why we shouldn't decompose work too quickly into fine-grained features. Sometimes, epicentre design can lead us to the stripped down core very quickly, but on other occasions it's not so simple, perhaps because we are not yet sure what the core solution will be. The business value can be fulfilled in many different ways. Some might not even involve building software at all.

Our Jobs4U website might have all sorts of potential features. Candidates could post their resumes; they can search for jobs based on location, job title and salary bracket; we could add advice articles or offer feedback on CVs and covering letters. You can easily imagine getting down to user stories very quickly: As a job-seeker, I need to upload my CV in pdf format so it can be read by employers. As a job seeker I need to upload my CV in Word format because it is easy for me to do.

The valuable need is simple, however: finding a job. If we institute a single feature that achieves this value for all our users, all the others are redundant. There's no point in fretting about which format we can accept when uploading the CV may be unnecessary.

Tim Brown in his book Change by Design comments 'there may be a tendency … to simplify the [design] process and reduce it to a set of specifications or a list of features. To do so is almost invariably to compromise the integrity of the product on the altar of convenience.'

CASE STUDY: Dyson and the underlying customer need

James Dyson is an innovator and entrepreneur known, among many inventions, for his ballbarrow, airblade hand-dryer and most famously his vacuum cleaner.

In 1979, Dyson bought an expensive vacuum cleaner. It soon began to lose suction – in fact Dyson felt it was just moving the dust around the room. The dust in the room was so annoying that it reminded Dyson of an industrial sawmill where a cyclonic separator removed the dust out of the air.

For decades vacuum cleaner manufacturers had worked on making vacuum cleaners more effective. They tried to increase the revolutions per minute to make the suction harder. Dyson's insight was to look at the vacuum cleaner and decide that the bag was actually getting in the way. Thinking about the cyclonic separator he wondered how he could recreate that in a vacuum cleaner. A quick prototype later and Dyson knew he could.

Figure 16. A Dyson bagless vacuum cleaner

Then followed a difficult five years of building and testing prototypes. In spite of the mechanical success, Dyson was unable to get anyone interested. He tried to license the idea to vacuum cleaner manufacturers who all refused it – nobody could see past the status quo of a bag that collected the dust.

Eventually, Dyson went to a bank to take out a £900,000 loan in order to start manufacturing himself. The story is that the bank manager was uncertain. He took the prototype home to his wife and asked if she thought it would ever catch on. She looked at the bagless vacuum cleaner and declared it was the best thing she'd ever seen. Why? Not because of the increased suction power, but because of something everyone else had overlooked.

Dyson knew about making the vacuum cleaner more powerful, but it was the bank manager's wife who had pointed out that no-one likes changing vacuum cleaner bags. They cost too much. There's never a spare one when you need it. They are messy and ugly. The idea of seeing when something is full, holding it over the bin and just emptying it in one easy movement, suddenly seemed extremely attractive.

Dyson found the underlying customer need almost by mistake. But he did it the right way – by creating prototypes and testing them. A more effective way in the end than years of expensive market research – which after all, every other company had been doing without uncovering the real customer need.

Activity 8: What's critical

This activity works best when you gather a small group from around the business who are all involved with or who have a stake in a particular project. The activity will take 30 minutes.

Provide some Post-it notes and ask everyone to think and then write out their 5 most critical requirements for the project. This should be done silently, with no conferring.

After 5 minutes, ask everyone to compare their critical requirements. How much do they overlap? Are there any surprises?

5.4. Elaboration: the conversation

In the XP ‘Card, Conversation, Confirmation’, elaboration might loosely be termed the conversation. A conversation fleshes out the details, provides the background and colour to the story. A conversation may not be enough – there may be all sorts of additional tools which have provided further information or notes on where to access further details – but the essence of the idea is as simple as two people talking about what is needed or wanted.

Who needs to be present? At a bare minimum, the person requesting it and the person expecting to deliver it should be there. In practical terms, on most teams this tends to mean a developer and a product owner. The meeting can sometimes include testers, architects or other specialists, depending on the work. These need not always be technical specialists, often users might be present, for example. Since it is a working meeting and the ‘idea’ represented as a story or feature may still be vague, other members of a customer team or stakeholders might also be present.

When should it happen? Trying to give the working week or month a rhythm often means that conversations about user stories tend to be held as part of a work planning session – perhaps in a formal sprint planning meeting, or before the team settle to a new batch.

In either case, the crucial point is that the conversation should not happen too far in advance of the actual work – this is ‘just in time’ elaboration. It’s important because of our fallible, human memories. Even with detailed notes, we find it hard to recall exactly what was meant in a meeting that happened three weeks ago. But also, the essence of Agile requirements is that we develop our understanding based on what has just happened. If last week’s demonstration uncovered a mass of feedback about the user interface, then we will want to apply this new learning to the requirements for the next set of features. Knowing that our ideas about such matters will change throughout the project means there’s little point in defining how we want everything to look in advance. It’s just wasted time.

What sort of meeting is it? Conversations are not monologues; elaboration meetings are not a series of instructions issued to a developer. This is a working meeting in which ideas may change and should be debated. Is that really what the customer wants? What might that look like? What else would it be nice to know? How could we test this? It’s also a time to ask about performance criteria around scalability, security, etc. The team might query the size of the story – whether they need to make it smaller in order to fit it within the next sprint or be able to predict its effort more accurately.

This is also the meeting in which the team will define what success will look like; how the team will know when the story is completed.

What do we do with the new information generated?

Capturing this information and any decisions is as simple as a developer taking notes – sometimes scribbled on the back of the card. The notes should act as a reminder, helping people recall what was said. In most cases, the code itself becomes its own documentation. Since a customer representative will be feeding back on the working software within a few weeks, what need is there to record changes to requirements in a more formal system?

In some cases, safety or security might mean an organisation requires greater traceability: when exactly did we decide to downgrade our security checks; what was our reasoning for doing so? Yet even then, the traceability needs to be seen in context – often our purpose would be better served by creating additional feedback loops to catch mistakes rather than in creating a paper trail.

There are lots of downsides to excessive documentation, but there are times when a certain level is useful. There is some information that we need to share more widely. If we have a project working at scale then some information from the conversation may need to be shared outside the room, and the code in itself may not be sufficient. Training guides, architectural decisions, the most critical requirements, intentions for the future… such valuable pieces of information are sometimes not best captured solely in the code. The team may use a discussion thread, a wiki or other method of commenting.

Any documentation for storing information needs to be useful and used. If it is not, then it is simply wasting storage space (whether that's paper on a shelf or bytes on a hard-drive). We try to reduce our dependence on documentation by being disciplined about the state of our codebase, the tests that show what it is doing and monitoring of faults and customer usage.

5.5. Story mapping

An important tool to assist the conversation and intended to help deal with some of the problems described in decomposition, is the story map. We explored how to create a story map for a customer journey in detail in the Delivering Early and Often session. We described it as a strand of functionality made of several different features that will provide real use and value to the customer.

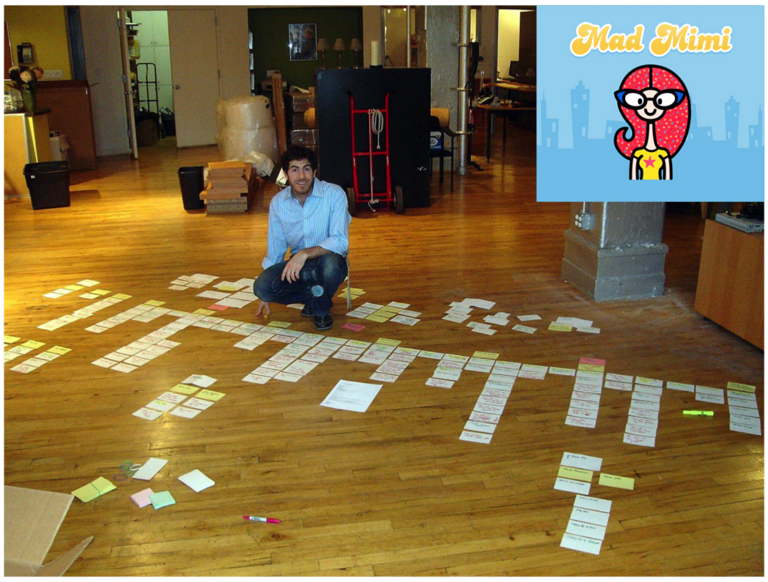


Figure 17. Gary Levitt, owner & designer of Mad Mimi, doing user story mapping

In essence, a story map helps you see how different features are connected. On the horizontal axis, the map shows a flow that mimics the usage order of a series of interactions (registration, selection, purchase, etc.). The vertical axis shows 'criticality' which may be either value or urgency. By drawing lines, the team can group together features according to different parts of the journey and also decide minimal subsets of functionality that still offer an end-to-end experience.

An alternative use of the vertical is to break down large features into smaller ones (rather than just grouping features by usage, the map can show how epics relate to stories, etc.).

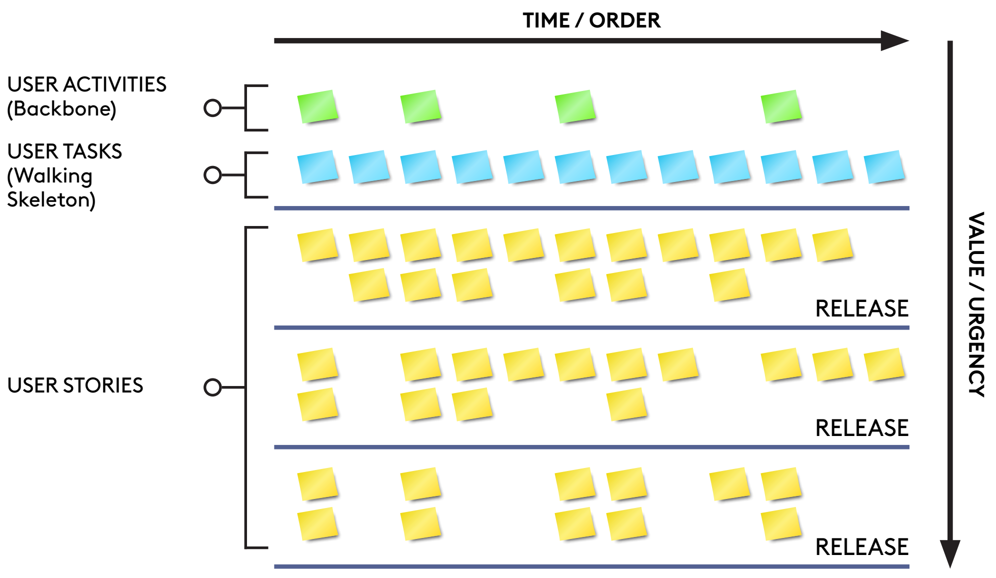


Figure 18. A typical story map structure

This tool is also very helpful as we work with requirements, allowing us to see how they fit together and emphasising gaps in our system or flow. A story map is a great forum – we can gather a range of stakeholders in front of the board (or wall or even room!) and ask them to check there are no glaring omissions in the usage as described by the map. This is also a way to show stakeholders who feel their requirements are being de-scoped why other features have priority and what alternatives or workarounds may be necessary until a further release.

As a visual tool, it helps give everyone a sense of the product as a whole, and keeps in mind how the user(s) will interact with it. The connection of larger stories to smaller ones allows us to continue connecting features to business value – something we hope should stop us losing the value as we decompose into ever smaller stories! Like any visual tool, it provides a shared reference point, which, with luck, should increase chances of a shared understanding.

5.6. Confirmation

In Alice in Wonderland, Alice is given some invaluable advice by the King of Hearts:

> "Begin at the beginning," the King said, very gravely, "and go on till you come to the end: then stop."

All user stories need boundaries, specifically something that tells us when they are completed. These criteria need to be explicit: to tell us when we have done enough, and the feature is working. Of course, we will still need to show it to a customer to check we are right, but this 'confirmation' stage, should ensure we're not wasting their time with something not fit for purpose.

Acceptance criteria

The most commonly used way of explaining what confirmation means it to write a series of acceptance criteria. These can range from an incredibly simple single line, to a whole series of statements which must be true in order for the story to be considered done.

They have a secondary purpose. Because they often explain what a completed story might mean – that is, how a feature is expected to behave or work – they also provide the necessary detail to the story. In many cases, the conversation stage described above involves asking questions and agreeing answers. From this information, the developer and product owner will jointly write the acceptance criteria. These then, become the documentation and notes that a team needs to write the actual code and represent a shared understanding of what a feature should do.

Each statement must be testable. This does not mean it needs to be written as code (although acceptance criteria can be turned into automated tests), simply that it is possible to check whether each statement has been satisfied or not.

Acceptance criteria are usually written in simple, everyday business language. To return to our fictitious Jobs4U example, a feature about uploading a candidate's CV might include the following acceptance criteria:

- A candidate can only upload 1 CV at a time.
- A candidate can upload a CV as a Word document or a pdf.
- A candidate will receive a message when the CV is successfully uploaded and saved.
- The system should be capable of uploading up to 100 CVs simultaneously.

Activity 9: The big balloon test

This activity needs a group of people who have never played it before – about 6-10 is ideal.

You need several packets of balloons, some marker pens, a ruler and paper.

Before the game begins, blow up a balloon and carefully draw a face on it. Make sure you have set out exact measurements – for example, eyes must be 5cm apart; mouth must be smiling, etc. You will use this to define the acceptance criteria (although you won't share these).

The game:

Show your balloon to the team. Explain you'd like them to make as many balloons as possible like this one within the next 2 minutes. You can answer questions if people have them. They are unlikely to question the precise measurements of the face or shapes, even if they ask about colour or design. After a couple of questions, start the timer.

At the end of the 2 minutes, ask someone to bring up the balloons. Simply accept or reject them according to your criteria. Normally, almost all the balloons will be rejects – and thus wasted work.

If people are annoyed or surprised gently remind them that this can happen in software development occasionally. Give them 2 minutes to decide how to improve matters before trying again. They should ask what are the acceptance criteria behind the accept or reject decision. You can now tell them exactly what is wanted. Give them another minute to decide on how they will achieve this.

Start the timer for another 2 minutes. At the end, accept or reject the work.

Discuss:

Hold a 10 minute discussion in which you talk about defining and clarifying acceptance criteria up front and about creating tests in advance that ensure you'll be passing them. If the team didn't think of it, point out that they could have made a paper template to draw the face or to check quality.

Specification by example

Rather than writing a series of statements, the acceptance criteria can also be viewed from differing user points of view. This specification by example is a more imaginative approach which can help to flesh out the story by making it clear to the team how the story will be used in a realistic environment. As a process, thinking of all the ways a user might interact with the story can catch edge cases and unusual interactions that cut down on problems later.

Naturally, coming up with the examples tends to be collaborative process. One person is unlikely to be able to think of all the ways a customer might interact with a payment system or their account. A small team of business analysts, developers and a domain expert (perhaps someone from customer service) come together to brainstorm what the key examples might be.

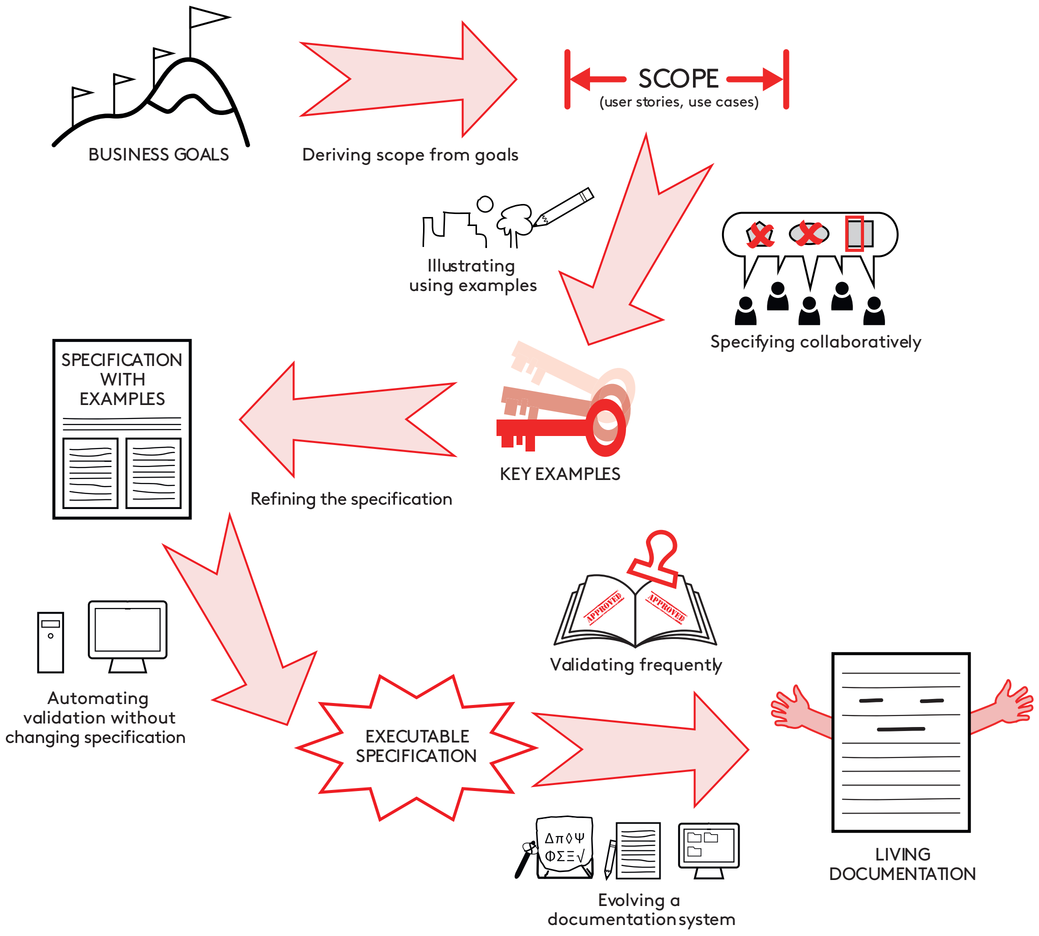

BUSINESS GOALS
Deriving scope from goals
SCOPE
(user stories, use cases)
Illustrating using examples
Specifying collaboratively
SPECIFICATION WITH EXAMPLES
Refining the specification
KEY EXAMPLES
Validating frequently
APPROVED
APPROVED
Automating validation without changing specification
EXECUTABLE SPECIFICATION
LIVING DOCUMENTATION
Evolving a documentation system

Figure 19. Key process patterns of specification by example (adapted from Specification by Example by Gojko Adzic)

For example, the team for an online delicatessen might be discussing card payments. Almost all credit and debit card numbers are 16 digits long, presented in four groups of four. What could be more natural than for the developer to create a form that mimics this grouping for customers to enter their card numbers? But the customer service manager, sitting in on the discussion, is concerned. After the last marketing campaign at the world food fair, there was a rush of orders from customers holding cards with overseas banks. Many of these cards have 17 or 18 numbers. The team writes these down as example cases which the final design will have to be able to cope with.

Have you ever stayed in a hotel without an alarm clock? Since you have a hugely important meeting in the morning, you decide to use the hotel's bedside telephone alarm. You're not quite sure you understand the instructions, so you test it by setting an alarm for 2 minutes time. The alarm sounds like a clarion. Perfect. You reset it for 7am and go to sleep confident that it will wake you up. Why are you so confident? You haven't tested that the alarm works at 7am! For all you know the alarm might have a special glitch that stops it working at that precise time. But rather than stay up all night testing the alarm constantly, you decide there is more value in getting a good night's sleep.

In his book Specification by Example, Gojko Adzic recommends using example tests that will make the problem concrete and prove you have solved the problem. Of course in an important system, developers would probably run a few more tests than our single alarm clock check. There will probably be one particularly quirkily minded tester who will come up with some truly bizarre edge cases to test. This is the kind of tester who would wait up until midnight to set the alarm clock in case it worked on a 12-hour, not a 24-hour clock.

Our point is that you must make a judgement call as to how hard you will look for edge cases. There is a risk of doing too much thinking up front, when a feedback loop could resolve issues for you more quickly.

A natural next step is to refine the acceptance criteria or acceptance examples so that they can be tested or automated very simply. Several approaches are available for this – and the most popular is Given – When – Then.

- Given some precondition
- When some action by the actor
- Then some testable outcome is achieved

Activity 10: Clarifying and refining your examples

This activity will take about 30 minutes of work, spread over a couple of days as you find people to help you test and refine your work.

Select a feature for a new project. Try to choose one about which you are unsure in order to reveal how far this technique can help you.

1. Write your initial specification with 3 key examples. Remember that you are focusing on what the system does, based on business value. You are not writing a script (which tends to follow a user's interactions with a system).
2. Checks: are you clear on what your example is specifying? Could it be specifying several things? If so, you need to refine. Does your example sound like a flow or a 'how' rather than a 'what'? If it does, refine.
3. Take your specification and examples to a colleague – possibly to someone who doesn't work on the project at all. Ask them to read it and tell you what it does.
4. If you find yourself having to explain anything, capture this and use it to refine your example or write it up as a 1-2 sentence introduction and put it into a heading.
5. Ask your colleague to imagine that they are searching for this specification on Google. What search terms would they use? Use this information to give your specification a descriptive, memorable title.
6. Now go back to your specification and re-write it in the Given – When – Then format. Keep things focused, you don't need to include every possible pre-condition in the 'Given', only those immediately relevant to this piece of business value. The wider context can be fixed in the automation testing layer, rather than at this specification level.
7. As a final check that you are using domain language consistently, show your specification to a non-technical person. Do they understand it? Check what the title and each action means to them.
8. Now take your specification by example and sit down with a developer and a tester. Try and think up some edge cases that might break the rules. Ask if you are missing anything. Do your examples illustrate particular areas of concern for them?

Commentary:

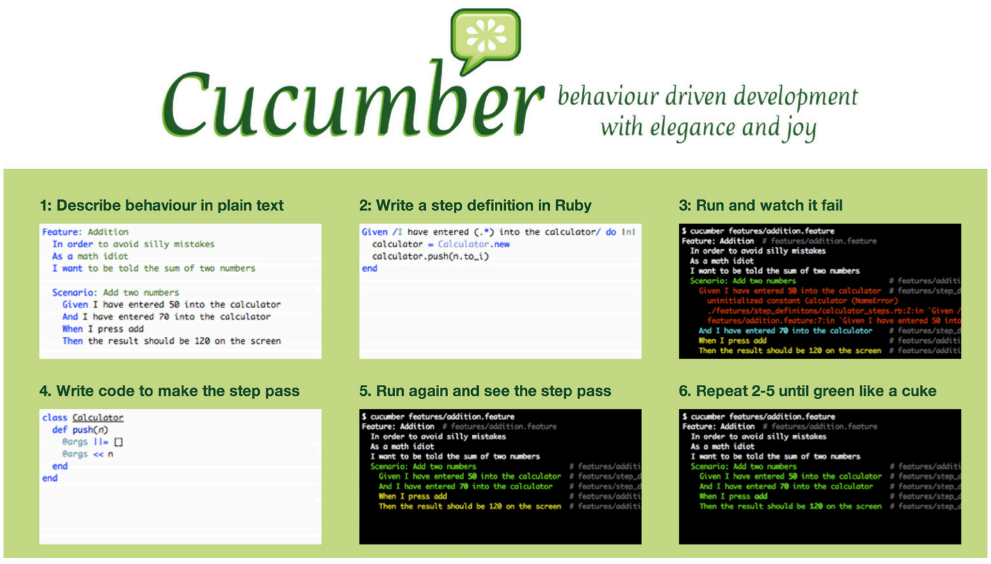
You should now have a much more precise specification with helpful examples which are testable. It might seem like quite a lot of work when we have spent so much time insisting on getting on with things and refining as you go, but remember that this exercise has deepened your understanding, has built in collaboration with others and has left you with something short, simple and self-explanatory that you or someone else can refer to in 6 months time and still know what was intended.

Executable specifications

It is not a large step from the syntax above, to go on to reframe the whole requirement in a kind of code that expresses the need, the expected set-up and the tests. The idea is that the whole requirement now forms a 'living documentation', intelligible and useful, to be shared by both technical and non-technical people. Adzic comments that 'If the key examples are easy to understand and communicate, we can replace the requirements with the key examples'. These can be extended into automated validation tests, stripping out yet another typical layer of mistranslation and 'the key examples stay almost in the form they were in on a whiteboard – human-readable and accessible to all team members.'

Doing this significantly reduces the need for further documentation (which is time-consuming and dull, rarely used and takes up space). It also helps fill the miscommunication gap that can exist when the requirement stops being a Post-it note written in English and is translated into code. While collaborative working over the card-conversation-confirmation stages should have reduced the gap, an executable specification should reduce it still further. On large or complex projects, executable specifications can prove an invaluable way to ensure everyone is working from the same shared understanding.

There should be far fewer 'but that isn't what I wanted' moments from the business owner or customer, and any changes that need to be made to the requirements are made in one place only – in a way that enables everyone to have greater visibility of the likely impact of change.



1: Describe behaviour in plain text

```
Feature: Addition
  In order to avoid silly mistakes
  As a math idiot
  I want to be told the sum of two numbers

  Scenario: Add two numbers
    Given I have entered 50 into the calculator
    And I have entered 70 into the calculator
    When I press add
    Then the result should be 120 on the screen
```

2: Write a step definition in Ruby

```
Given /I have entered (.*) into the calculator/ do |n|
  calculator = Calculator.new
  calculator.push(n.to_i)
end
```

3: Run and watch it fail

```
$ cucumber features/addition.feature
Feature: Addition  # features/addition.feature
  In order to avoid silly mistakes
  As a math idiot
  I want to be told the sum of two numbers
  Scenario: Add two numbers                     # features/additi
    Given I have entered 50 into the calculator  # features/step_d
      uninitialized constant Calculator (NameError)
      ./features/step_definitons/calculator_steps.rb:2:in `Given /
      features/addition.feature:7:in `Given I have entered 50 into
    And I have entered 70 into the calculator    # features/step_d
    When I press add                             # features/additi
    Then the result should be 120 on the screen  # features/additi
```

4. Write code to make the step pass

```
class Calculator
  def push(n)
    @args ||= []
    @args << n
  end
end
```

5. Run again and see the step pass

```
$ cucumber features/addition.feature
Feature: Addition  # features/addition.feature
  In order to avoid silly mistakes
  As a math idiot
  I want to be told the sum of two numbers
  Scenario: Add two numbers                     # features/additi
    Given I have entered 50 into the calculator  # features/step_d
    And I have entered 70 into the calculator    # features/step_d
    When I press add                             # features/additi
    Then the result should be 120 on the screen  # features/additi
```

6. Repeat 2-5 until green like a cuke

```
$ cucumber features/addition.feature
Feature: Addition  # features/addition.feature
  In order to avoid silly mistakes
  As a math idiot
  I want to be told the sum of two numbers
  Scenario: Add two numbers                     # features/additi
    Given I have entered 50 into the calculator  # features/step_d
    And I have entered 70 into the calculator    # features/step_d
    When I press add                             # features/step_d
    Then the result should be 120 on the screen  # features/step_d
```

Figure 20. Screenshot from the cukes.info website

The only risk is that the team can become overly involved in thinking up the tests in advance – once again leading to overly heavy specifications and falling into the old trap of forgetting how much of software design is discovered in the doing rather than the planning.

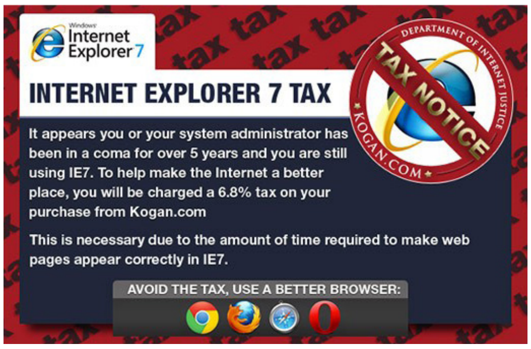
You may hear this technique referred to by differing names, including Behaviour Driven Development or occasionally even Acceptance-Test Driven Development. There are small semantic differences between the use of such terms, but the essence of the idea remains clear: create a shared, intelligible language that sets out the intention and the examples or statement that will prove it and provide an underlying series of automated tests.

CASE STUDY: Kogan challenges acceptance criteria

Many acceptance criteria include details on which platforms the software should work on. A typical list might run: Chrome, Safari, Firefox, Internet Explorer. It would also include the versions that the software must work for. Sometimes such criteria are handed out with little thought for the impact they will have on development time and cost, or with any consideration of how valuable they are (that is, how many users there are for each).

The Australian company Kogan (an online electronics retailer), decided to pass on the cost associated with supporting different browsers to the customer. Internet Explorer 6 and 7 were both out of date, but they had customers who still used them, about 9.5% of the web population.

According to Kogan, making the website look normal on these outdated versions cost the web team in time and effort. The decision was taken to ignore these edge cases, or rather, to pass on a charge that reflected the cost required to include them. As one reviewer grumpily commented, 'anyone buying from the site who uses Internet Explorer 7 will be lumped with a 6.8% surcharge - that's 0.1% for each month Internet Explorer 7 has been on the market.'



INTERNET EXPLORER 7 TAX

It appears you or your system administrator has been in a coma for over 5 years and you are still using IE7. To help make the Internet a better place, you will be charged a 6.8% tax on your purchase from Kogan.com

This is necessary due to the amount of time required to make web pages appear correctly in IE7.

AVOID THE TAX, USE A BETTER BROWSER:

Figure 21. Tax notice for using older versions of Internet Explorer

Most companies are unable to pass such charges on to the customer. But it is certainly worth considering whether acceptance criteria should be challenged if they specify this kind of time-intensive extra work. Refusing to support nearly 10% of your potential customers might sound drastic, but if it makes significant time and cost savings then it might be precisely the right decision to take. No matter what the eventual choice, having the conversation as an entire team will mean coming up with acceptance criteria whose impact is properly understood.

6 SUPPLEMENTARY TOOLS FOR AGILE REQUIREMENTS

It is possible for templates and tools to restrict your thinking. It would be foolish to claim that the basic card-conversation-confirmation stages that we have described above are always sufficient. There may be all sorts of other pieces of information or alternative ways of presenting the information that assist you in building a system. We include some of the most popular of them here.

The key point to remember is that you should not spend excessive amounts of time on these tools. They are just that – tools – not products in their own right. Be careful not to use an elaborate model or data diagram as an excuse for not getting on with building something and learning more about it through feedback.

6.1. Agile modelling: data diagrams and UML

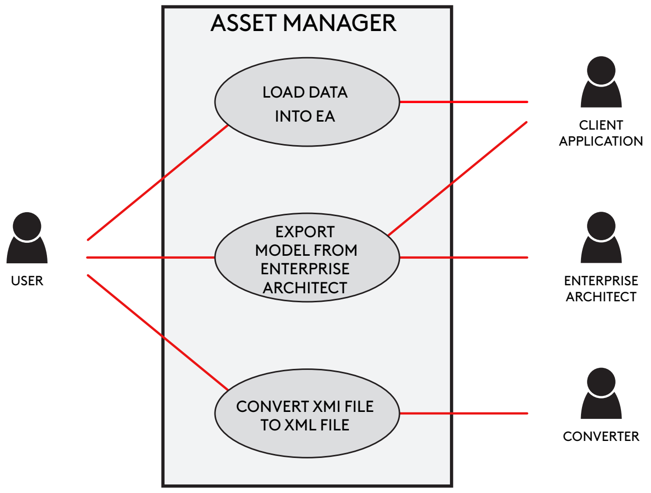


Figure 22. Example use case diagram

Scott Ambler describes ‘agile modelling’ as the process of walking a tightrope between too much detail that will weigh you down and that which will help you work more effectively. Used well, models can assist in design by helping you think about and explore possibilities for the system. They can also help communicate ideas to others to enable a better quality conversation. A data model might help reveal the structure of the database to those who are going to write applications that interact with it. Diagrams might be helpful to show the different business processes the system will connect, while wireframes might help explain the overall flow of information and the functionality. There are dozens of different types you could use, whether you draw them up on a white board or write them in UML (Unified Modelling Language).

Unified Modelling Language was developed to provide a standard way of building a visual representation of a system's architecture. It offers a static depiction of the system's structure, normally the architecture, including objects, attributes, relationships and operations. At the same time, UML can be used to look at the changing interactions within the system and its behaviour (sometimes called the dynamic view).

Doing this is often very helpful, allowing you to map out how a system is at the moment, and how you might want it to be. However, there is always a danger of becoming enamoured with one's own models and with spending too much time on perfecting them in advance. The more effort that is put into blueprints and UML, the more risk there is of this. Of course, it can be very convenient to have a shared vocabulary and point of reference, but this can risk us assuming that the model in itself has longevity or value when it is simply an aid, to be discarded swiftly.

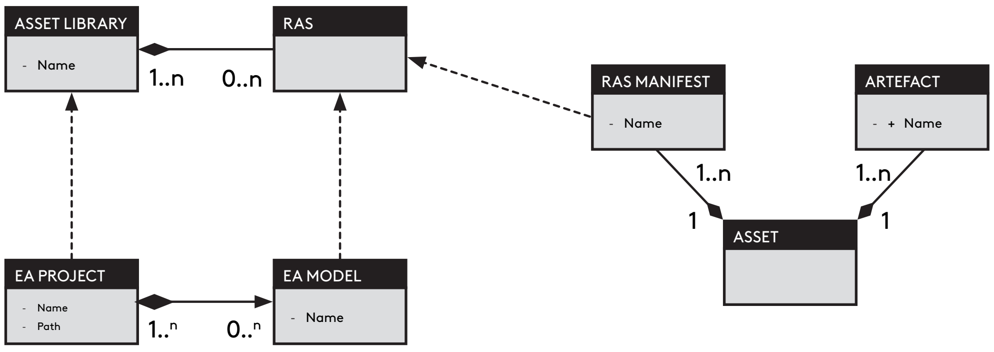


Figure 23. Example of an entity relationship diagram

6.2. Use cases

A use case is defined as the set of interactions required between a user (or 'actor') and a system in order to achieve a goal.

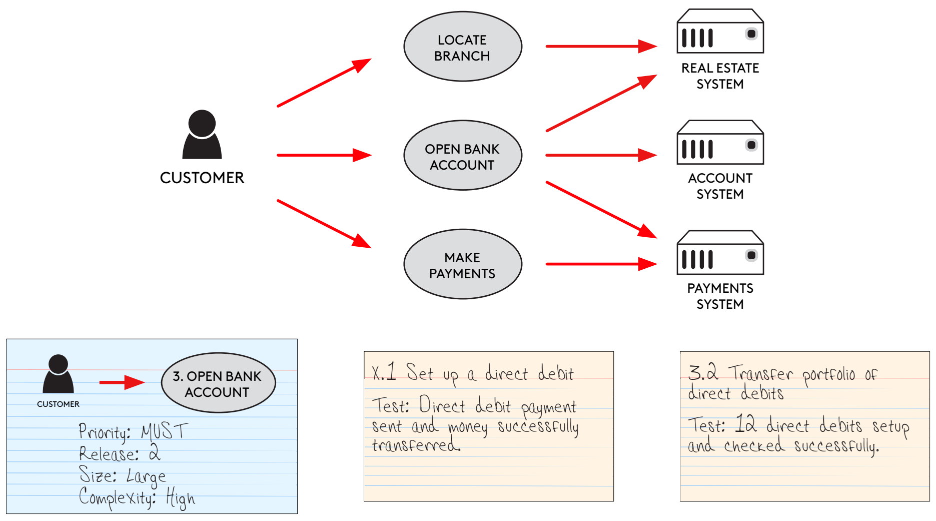

LOCATE BRANCH
OPEN BANK ACCOUNT
MAKE PAYMENTS
CUSTOMER
REAL ESTATE SYSTEM
ACCOUNT SYSTEM
PAYMENTS SYSTEM
CUSTOMER
3. OPEN BANK ACCOUNT
Priority: MUST
Release: 2
Size: Large
Complexity: High
X.1 Set up a direct debit
Test: Direct debit payment sent and money successfully transferred.
3.2 Transfer portfolio of direct debits
Test: 12 direct debits setup and checked successfully.

Figure 24. Example use case diagram showing actors within the system

Use cases have fallen out of favour. Perhaps because they used to be symptomatic of heavy specification and documentation: a collection of sequence diagrams, class diagrams, etc. laid out for every single potential use or behaviour of the system. This information was all potentially useful, but it suffered from all the problems we've already discussed: it could take so long that the information was already decaying; the detail actually inhibits conversation, and it doesn't encourage learning in the process of writing code because it specifies everything up front.

However, there is a role for a more Agile version of use cases to help you evolve your design by exploring ways the system could be used. A sketch of the system can help keep sets of possible interactions related at a higher level than the usual story or requirement level.

Imagine that you are drawing out all the ways people might want to interact with a property auction website. You will have buyers, sellers, agents and employees to consider – these are your 'actors'. If you decide that buyers are the most important user, then you can work on examining this case in more detail, comfortable with leaving the others. Later, you can return and begin to flesh these cases out as well, or decide that you are not going to bother because you have already captured most of the value. In other words, use cases can be a tool to help you deliver even your requirements incrementally.

Ivar Jacobson calls this revised way of looking at the tool 'Use-Case 2.0' and he identifies six principles which embody this lighter and more scalable type of use case.

1. Keep it simple by telling stories

Essentially this can help map our user stories onto one another so that we see how they interact. The use case does not make the stories redundant, it simply helps show how they relate to one another.

2. Understand the big picture

This can also be done at a less granular level – before you have split ideas down into user stories, when you are simply asking bigger questions about the kind of things you want the system to achieve. This helps you draw out the scope and boundaries for the project in a way everyone can grasp easily. Providing this overview diagrammatically can be helpful in clarifying how stories connect to the critical requirement as well.

3. Focus on the value

The use case should focus on how the system will be used. It describes a desired behaviour rather than intent. This may sound like a semantic difference, but thinking how the user is most likely to interact with the system helps us prioritise. The usual interaction comes first, then you note down all the alternative interactions. Some call this 'feature injection', always focusing on the part of a system that will be useful and writing it as 'In order to...' e.g. in order to access files remotely, the user logs on to the app.

4. Build the system in slices

Identifying the most valuable interaction also means we can build all of this interaction as an end-to-end 'slice'. This could be broken down into individual user stories. A user story should still provide functionality to users and as such is still a slice through processes not just a step, (see elephant carpaccio in the Delivering Early and Often session). Jacobson points out that each 'slice' must contain its own acceptance tests – essentially providing all the documentation or traceability required.

5. Deliver the system in increments

You have selected the most valuable and central interaction as a 'slice' through the system. That's your first increment. Actually, depending on what level you have drawn the system at, you may want to sub-divide again. If the initial slice is 'access sales data remotely' you may only want to include a particular region. Perhaps your mobile sales team are the ones who need the feature and thus only sales-related files for the North American region will be relevant.

6. Adapt to the team's needs

This should probably go without saying. Some projects will need more detail than others. That's fine, but strive to keep the use case lightweight. Your system sketch is a tool to help you deliver software, not a work of art to be displayed in a museum.

7 NON-FUNCTIONAL REQUIREMENTS

Almost everything we have talked about up until now has been related to functional requirements. When we begin to look at non-functional requirements we open a whole different can of worms. They describe system attributes or constraints and include security, reliability, maintainability, scalability, and usability.

They are crucial to how the user experiences the product and how the organisation runs it. You may have created the most superbly intuitive, beautiful and useful application imaginable, but if it collapses when more than three people try to use it at any one time, then it is of no practical use. If you built it for peanuts and the CFO loves you, he will love you a lot less when it turns out to cost millions to maintain. If the app has incredibly cool flashy graphics but these can only be accessed on a bandwidth far in excess of anything going to private houses… you get the picture. Systems must be not only fit for purpose, but fit for use. For example, the success of the AK47 rifle as an assault weapon for guerrilla armies everywhere, is less about its accuracy or firepower, than its performance when poorly maintained and its simplicity – allowing it to be used by fighters with a bare minimum of training.

Most software project failures are not about a lack of features, but because the product does not perform in the way customers wanted it to. The software might work but not be fast or reliable enough. It might be brilliant but be so difficult to use that hardly anyone can be bothered to switch over to it. What really matters is a product's qualities and what our customers and stakeholders value – and both of these connect to the non-functional elements of a product as much as to the functional. In spite of this, non-functional requirements (NFRs) have a tendency to be – not forgotten about, exactly – but left vague and ill-defined. In some cases, however, they are completely over-specified as part of a just-in-case mindset.. Sometimes, rather than being considered depending on the needs of a particular project, they are simply assumed to be the same as for the last project. Whether under or over-specified, the result can be highly damaging to the final product.

If NFRs have been written in advance and handed to the development team another problem can arise. It is possible to make these poorly defined requirements a convenient catch-all for architectural constraints and design decisions. This allows them to become a sneaky way of fitting large amounts of up-front design work into a project.

Consider the following innocuous-sounding requirements:

"Help text will be provided in English, French and German". This contains functionality and incorporates a great deal of work based on concealed assumptions about what is necessary.

"Up to 300 users may be using the update customer at any one time". Once again this conceals an assumption which will drive work and cost – we will need to buy servers to deal with this concurrency, for example. It may be entirely wasted cost.

We've offered just two examples. In reality a NFR list could contain hundreds – often done up front, littered with hidden assumptions and concealing large amounts of work. Thus, they tend to work against incremental and iterative delivery, encouraging up front design and increasing batch size. If the NFRs lack clear success criteria and simply state a system should be 'fast' or 'user-friendly' then they are even more likely to drive large increments because the work is unbounded.

Activity 11: When is an NFR not an NFR?

This activity can be carried out on your own and should take about 20 minutes.

Look at the following table. How many of them would actually be solved by creating new functionality and therefore should be re-written as normal functional requirements? How many of them contain hidden assumptions which should be exposed in order for us to test their validity?

NFR	Re-write?	Assumptions?
All comment fields must be spell-checked		
Customer data can be exported in XML format		
Changes required by law will be applied at least 3 months before the law becomes enforceable		
The customer help desk will support users between 8am and 5pm on weekdays, excluding public holidays		
Up to 200 new sites per year may start to use 'update customer'		
If the user cancels or quits 'update customer', all changes made by the user will be reversed		
For American users all dates will be displayed in the MM/DD/YYYY format		
All data will be backed-up daily		

Commentary:

When you go through lists of NFRs with your team on a project, it can be a good exercise to think about them in the same way – teasing out hidden assumptions and pieces of functionality. Remember the point of this is not to hold up the NFRs as badly written, but to see how you can understand them better in order to improve the design and solution for the customer and user.

7.1. What are the real non-functional requirements?

"A hint of the software industry's immature understanding of product development, is the word used for Stakeholder Values and/or Product Qualities, Non-Functional Requirements. The words Non-Functional says something about what the Requirement is not, but nothing about what it is. It's like describing a friend as a non-fish. While that is probably a true statement, it is not informative.

... Whenever I see the words non-functional requirements in my clients' documentation, I immediately know that they do not have control over Stakeholder Values or Product Qualities. Lacking that control, they don't have control over costs either."

Tom Gilb

We've said that many requirements called non-functional, are actually functional requirements in disguise. Nonetheless, there are unquestionably requirements which do not relate to functionality, but which are important. Gilb, as his quote makes clear, objects to the lack of clarity in the term non-functional requirements.

Gilb divides requirements into several categories. We're not suggesting that his definition is perfect or complete, or that you should use these sub-categories, but it does highlight that thinking more deeply about 'non-functional' requirements might be helpful to your organisation and project.

Requirements that constrain our system

- **Performance constraints** – what level of quality do we want our product to perform at? How fast, secure, simple or accessible should it be? Do we want to access it 24 hours a day? Do we mind if it is a little slower at 4am?
- **Resource constraints** – what level of resource can we or are we willing to expend on this product? How much time do we have? How much cash can we spend to develop a product or to maintain it? What should our total cost of ownership be?
- **Design constraints** – pre-design decisions that must be explicitly incorporated in the up-front design, including architectural decisions such as SOA or using the Enterprise Service Bus for messaging.
- **Conditional constraints** – these constraints tend to capture decisions that will impact at a system level. For example, we might say: 'Internet Explorer 7 will not be allowed to access the application as we will not be providing support' or 'this application must be legal throughout Europe'.

7.2. How should we treat non-functional requirements?

The team need to apply the same criteria as for functional requirements:

- **Connection to business value** – each NFR must link to business value and then be negotiated based on what the business truly needs – how scalable does the system need to be? Are we building for use or for reuse? Just how important is the latest security?
- **Testable** – each NFR also needs acceptance criteria which can be tested in order to check whether features or the system as a whole are complying with what is expected of them.
- **Sized appropriately** – each NFR should be captured at the right size or level; that is, where differing parts of the system have different requirements, these should be broken down to the appropriate level of detail. It might not be important to offer high performance availability on support applications, for example.

The last point is especially important as part of incremental delivery. It might be possible to allow some parts of the system to work at a lower quality while the team improve and upgrade features one-by-one.

Most NFRs should be addressed using just-in-time selection (much as functional requirements are). For example, there is no point in developing the system's portability onto different platforms at the beginning of the project, long before there is much understanding of the user's needs. This isn't always as simple as it sounds. Determining whether the requirement relates to a solution or not has an impact on how late we can leave decisions. Hardware, for example, has a significantly longer lead time to purchase, deliver and set up; while a legal compliance requirement might also involve a chain of decisions and implementations which increase lead time. The ideal is to maintain options as long as it's sensible to do so, avoiding upfront design wherever possible.

Rather than being hived off into some special black box for IT, NFRs can be treated in much the same way as normal requirements when it comes to prioritisation. In fact, we suggest that separating the two processes, when they can usually be managed together, often leads to a lower quality solution or to excessive costs caused by an over-engineered architecture. In either case, an emphasis on delivering early and often will help.

7.3. The importance of scalar requirements

Performance constraints are almost always 'scalar'. They ask how well something performs, and this tends to be a relative term. A fast journey in 1816 meant using a coach and six horses travelling at 16 miles per hour. In 1916, it meant going by train, at a top speed of 50 miles per hour. In 2016, we can assume that airplane speeds will remain the same at approximately 600 miles per hour. In other words, we can never just say something needs to be 'fast', we need to ask how fast, and what we might be prepared to spend or risk to achieve this.

Figure 25. You will get there eventually but parking might be a problem

Right at the start of this, admittedly fairly lengthy, session we talked about how we often treat requirements as contracts, and the problems this causes us. Scalar requirements are a perfect example of why it's so dangerous – they don't act in a binary fashion, but on a scale. Words such as: 'improve', 'better', 'equal' or 'reduce' are signs that we can't treat the requirements as items to be ticked off on a list. Instead we must ask deeper questions – where are we now; what are we prepared to put up with; what's the best we could hope for?

Defining and measuring what we mean by 'user-friendly' or 'secure' isn't always easy, but it helps us resist the urge to apply pre-packaged solutions which may be lacking in true innovation. Remember the vacuum cleaner manufacturers who assumed that what customers wanted in a 'better' vacuum cleaner was 'more suction power', when in fact it turned out to be a very different characteristic.

In his book, Evo, Gilb gives an example of trying to define and measure 'user-friendliness'. If we had a new mobile phone, we could specify one particular element as a test of how user friendly the phone is: learning to use the contacts on a mobile phone. To know how user friendly we are now (our bench mark), as well as how user friendly we'd like to be (our performance target), we need to set a 'scale'. Here suggested as, the 'average time in minutes to learn how to program numbers into the phone's memory'.

Of course, this measure is only useful when combined with what Gilb calls the 'meter', the process of how we measure. In the example given it was, 'average time taken by five people who have not had a mobile telephone before, as well as five people who have'.

An Emergn view

In an Agile world, when we build functions, we tend to build tests of that functionality alongside. We need to do something similar with non-functional requirements. Since our non-functional requirements tend to be cross-cutting concerns, any change – including additional users, different hardware, etc. – may not only have an impact upon one function but upon series of functions. This means that tests of performance requirements are even more important than they are for normal functional requirements. Despite this, the majority of teams in software development ignore or put off setting up tests for non-functional requirements.

Scalar requirements can actually help us reinforce one of the key feedback loops – monitoring. Setting up our non-functional requirements tests helps us look at how our system is being used and how it is performing, to provide a 'warranty' for the system's future. The warranty defines not only what we want, but also what we have typically defined as being an unacceptable level of service. Within these parameters, the development team guarantee that the system will perform as requested. If performance drops, it is normally a sign that the original parameters may have changed.

For example, perhaps we had predicted our system was unlikely to have more than 3 million users and our performance was based on this. Several years later, success has brought us closer to 5 million users and we are beginning to see compromises in quality. The compromises act as an early warning system, prompting us to return to our original non-functional requirements and then update them. Obviously decisions made on the basis of 3 million users are no longer valid. We need to reconsider them in the light of the new reality of 5 million users.

The point of a warranty, we should point out, is not to wait until it is broken, but to point out when we are approaching certain pre-defined limits. Just as a fuel gauge goes red when the tank is nearly empty, rather than waiting until the car stops, we should use approaching a warranty to trigger an alarm. Automated testing alerts service management teams that there is a need to return to the system's performance requirements, update them and then set a schedule for improvements. This means the long term health monitoring of our system can be built into the solution itself – long after the original development team has stopped working on the project.

8 CONCLUSION

We've said that requirements are pieces of information. As we know, most information contains assumptions. The most dangerous assumptions are the invisible ones. In IT, for example, as soon as we hear about a business problem we start thinking about how we can solve it – through IT. Well, yes, you say, that's what we do. It's normal human behaviour – ask any specialist how to solve a problem and they will apply their specialism. It's so common that entrepreneurs warn against having any specialists in start-ups because they will push development into their own particular area. Equally, rather than demanding 'a new system' or 'tool', the onus is also on business stakeholders to state the desired outcome or problem and allow a more creative and innovative solution to emerge through collaboration.

Business problems that are brought to IT do not always require an IT solution. The assumption that they do can be a very dangerous one and lead to truly monumental waste. We should always be looking to uncover the assumptions in information – to check that what we are doing is genuinely valuable. That's why any session on requirements is also a session on gathering feedback and moving fast.

Trying to specify everything at the beginning of the project by 'building the perfect requirements' is not just impossible, it's absurd. We find out what matters as we progress, so where we start is not actually as important as starting. We should never delay in making the solution tangible – in many cases that means creating a kluge so that we can discover which bits to fix through testing and iteration. The first tangible solution may not even be an IT one – we might do things manually until we know we've got the right idea and then build the IT system later.

Learning outcomes

Now that you have completed this session you should be able to:

Identify how the term requirement is used in projects today

- Requirements gathering and formulation is regarded as critical to the project's success or failure
- The term requirements suggests they are mandatory, an exact depiction of the customer's needs or wants

Explore problems caused by the conventional approach to requirements, particularly the impact of separating design from delivery

- Getting requirements 'right' involves significant expense and up-front design
- The risk is that up-front design will be reworked (increasing waste) or will be retained and constrain the solution (reducing potential value)
- Too many requirements obscure the true value

Appreciate the consequences of seeing requirements as information

- Not all information is equal and there are significant failures in information quality
- Information has a lifecycle and its value decays over time
- Information is growing exponentially – this can overwhelm action

Judge how Agile practices differ in their conception of a requirement

- Agile attempts to welcome changing requirements
- It tries not to over-invest in up-front design or specifications, but to capture information quickly and cheaply, and communicate regularly
- Understand the Card, Conversation, Confirmation aspects of the requirements process

Understand and apply differing Agile techniques – select when each is most appropriate and appraise their advantages and disadvantages.

- User stories:
  - Low cost way to capture and store information
  - How to write effective user stories including using the INVEST model
  - Potential problems, limitations and errors with user stories
- Decomposition:
  - Using decomposition to assist in incremental delivery
  - Ways to split work
  - Decomposition at scale
  - Problems with decomposition
- Feature injection:
  - Preserving the link with business value
- Critical requirements:
  - Identifying and prioritising the critical requirements
- Elaboration:
  - How to elaborate through conversation and other tools
  - Capturing and using new information
- Story mapping
- Confirmation
  - Understanding and using acceptance criteria
  - Understanding and using specification by example and executable specifications
- Supplementary tools
  - Agile modelling
  - Use cases

Recognise the importance of non-functional requirements – discriminate when they are used improperly and distinguish how to use them better in the context of your project

- Giving due weight to the importance of NFRs
- Correctly defining and measuring NFRs
- Particular tools or considerations for NFRs

BIBLIOGRAPHY

Adzic, G., 2009. Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing. Neuri Limited.

Agile Alliance, 2009. Feature Injection: A Gentle Introduction. [online] Available at: <http://agile2009.agilealliance.org/node/185/>. [Accessed 23 August 2013].

Ambler, S. Agile Requirements Best Practices. [online] Available at: <http://www.agilemodeling.com/essays/agileRequirementsBestPractices.htm>. [Accessed 23 August 2013].

Appelo, J., 2010. Management 3.0: Leading Agile Developers, Developing Agile Leaders. Addison Wesley.

Artlyst. Sir James Dyson Biography. [online] Available at: < http://www.artlyst.com/James.Dyson>. [Accessed 23 August 2013].

Australian Skeptics. The Information Challenge. [online] Available at: <http://www.skeptics.com.au/publications/articles/the-information-challenge/>. [Accessed 23 August 2013].

Brown, T., 2009. Change by Design: How Design Thinking Transforms Organizations And Inspires Innovation. Harper Collins.

CNN Money, 2006. Top 10 Best Jobs. [online] Available at: <http://money.cnn.com/popups/2006/moneymag/bestjobs/frameset.7.exclude.html>. [Accessed 7 November 2012].

Cockburn, A., 2006. Agile Software Development: The Cooperative Game. 2nd Edition. Addison Wesley.

Cooper, A., 2004. The Inmates Are Running The Asylum: Why Tech Products Drive Us Crazy And How To Restore Sanity. 2nd Edition. Sams.

Dyson. Inside Dyson. [online] Available at: <http://content.dyson.co.uk/insidedyson/>. [Accessed 24 August 2013].

Early, M., 1986. Relating Software Requirements To Software Design. [online] Available at: <http://dl.acm.org/citation.cfm?id=12923.12924>. [Accessed 7 November 2012].

Floridi, L., 2010. Information: A Very Short Introduction. OUP Oxford.

Gasko, T., 2010. The Woman Behind the Vacuum Cleaner. [online] Available at: <http://www.vdta.com/Magazines/MAY10/fc-GaskoMay10.html>. [Accessed 24 August 2013].

Gilb, K., 2007. Evo: Evolutionary Project Management And Product Development. [online] Available at: <http://www.gilb.com/tiki-download_file.php?fileId=27>. [Accessed 24 September 2013].

Gilb, T., 2005. Competitive Engineering. Elsevier Butterworth-Heinemann.

Gottesdiener, E., 2002. Requirements by Collaboration: Workshops for Defining Needs. Pearson Education.

Gottesdiener, E., 2005. The Software Requirements Memory Jogger. GOAL/QPC.

Hass, K., Wessels, D., Brennan, K., 2007. Getting It Right: Business Requirement Analysis Tools and Techniques. Management Concepts.

Helm, B., 2012. How I Did It: James Dyson. [online] Available at: <http://www.inc.com/magazine/201203/burt-helm/how-i-did-it-james-dyson.html>. [Accessed 24 August 2013].

IIBA, 2009. A Guide To The Business Analysis Body Of Knowledge (BABOK Guide). 2nd Edition. International Institute Of Business Analysis.

IIBA. History Of IIBA. [online] Available at: <http://www.iiba.org/IIBA/About_IIBA/Governance/History_of_IIBA/IIBA_Website/About_IIBA/Governance_pages/History_of_IIBA.aspx?hkey=7f36c8c0-28da-468b-b9da-bc4e1077d69f>. [Accessed 7 November 2012].

Leffingwell, D., Widrig, D., 2010. Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise. Addison Wesley.

Marcano, A., 2011. Old Favourite: Feature Injection User Stories on a Business Value Theme. antonymarcano.com, [blog] March 24. Available at: <http://antonymarcano.com/blog/2011/03/fi_stories/>. [Accessed 23 August 2013].

McManus, J., Wood-Harper, T., 2008. A Study In Project Failure. [online] Available at: <http://www.bcs.org/content/ConWebDoc/19584>. [Accessed 7 November 2012].

Mills, G., 2011. New Benchmark Study on Strategies for Project Recovery Reveals the Average Firm Risks Losing At Least $74 Million in Failed Projects Each Year. [online] Available at: <http://www.pmsolutions.com/news/view/new-benchmark-study-on-strategies-for-project-recovery-reveals-the-average>. [Accessed 7 November 2012].

North, D., 2009. The Perils Of Information. Dan North & Associates, [blog] 1 July. Available at: <http://dannorth.net/2009/07/01/the-perils-of-estimation/>. [Accessed 23 August 2013].

Reinertsen, D., 1998. Managing the Design Factory: A Product Developer's Toolkit. Simon & Schuster Ltd.

Robertson, S., Robertson, J., 2006. Mastering The Requirements Process. 2nd Edition. Addison Wesley.

Snowden, D., 2012. Dark Satanic Mills & Sylvan Glades. Cognitive Edge Network, [blog] 22 October. Available at: <http://cognitive-edge.com/blog/entry/5752/dark-satanic-mills-sylvan-glades/>. [Accessed 7 November 2012].

Tschäppeler, R., Krogerus, M., 2011. The Decision Book: Fifty Models For Strategic Thinking. Profile Books.

The Standish Group, 1995. Chaos Report. [online] Available at: <http://www.projectsmart.co.uk/docs/chaos-report.pdf>. [Accessed 7 November 2012].

Wiegers, K., 2006. More About Software Requirements: Thorny Issues And Practical Advice. Addison Wesley.

West, D., 2011. Water-Scrum-Fall is the reality of Agile for most organizations today. [online] Available at: <http://www.forrester.com/WaterScrumFall+Is+The+Reality+Of+Agile+For+Most+Organizations+Today/fulltext/-/E-RES60109?al=0> [Accessed 7 November 2012].

Yatzeck, E., 2011. BDD, Feature Injection, and the Fallacies of Product Ownership. Pragmatic Agilist, [blog] 15 November. Available at: <http://pagilista.blogspot.co.uk/2011/11/bdd-feature-injection-and-fallacies-of.html>. [Accessed 23 August 2013].

Value
Flow
Quality

valueflowquality.com

emergn.com

by emergn