Why do requirements change?

> "Stable requirements are the holy grail of software development." (McConnell, 1993)

Once upon a time stable requirements where seen as a pre-requisite for starting a software development project. There may be a few Civil Servants who still belive this but many in IT world has given up looking for the Holy Grail of stable requirements[1].

Changing requirements have become an accepted fact of life for software developers, indeed, most of the process and methodology books now come with subtitles like "Embracing change". But how many of us stop and think about why requirements change?

I've been giving some thought to this question for a while now and I've come up with some reasons why I think requirements change. I'm not saying this is an exhaustive list but it is a list that makes sense to me based on my own experience and the way I view software development.

Why do we fail to capture requirements?

Perhaps the most obvious reason that requirements change is that we fail to capture them to start with. Someone writes down "black" when they should have written "blue." Everyone makes mistakes from time to time and a small mistake by a business analyst can easily go unnoticed for months. Sure we have document reviews to catch this kind of thing but such mistakes are easily missed in a 100 page tome.

There are lots of opportunities for mistakes in the requirements capture phase and not all of them are because some people are better than others. At first we need to comprehend the requirements then we need to capture them and communicate them, usually this is done with a text document. Mistakes can arise at and point: comprehension, recording or communication.

Any form of communication involves at least two parties: the sender and the receiver. Typically the business analyst will need to send their understanding of the problem to the developer (the receiver.) The important thing to realise is that the content of the message is decided by the receiver, it is they who interpret the communication and decide what it means. No matter how much effort the sender puts into their message they have no means of guaranteeing it is interpreted as they intend.

Now there are two opportunities for error here. We could assume that the receiver knows very little about the problem domain, to compensate we write a lengthy document that discusses all the details necessary. Unfortunately this approach risks overwhelming the receiver with details so they miss some of the important points.

Alternatively, we could assume that our developer knows quite a bit about the problem domain already and just communicate the bare essentials. The trouble now is that we are reliant on the knowledge the developer already holds, any omissions or

---

[1] That other Holy Grail "reusable software" may also be finding a few less devotees but that is another story.

errors in their knowledge will actually introduce changes which need correcting later on.

More subtly, the developer may have good knowledge about the problem domain with few omissions or errors but this may lead them to use assumptions and mental short-cuts which have worked well in the past but aren't appropriate in this case.

Developers aren't the only ones who may hold hidden assumptions, the same may be true of the business analyst, or even the end-users and managers who are commissioning the system. Few businesses have a written operating procedure, often the arrival of business analysts will be the first time someone has ever tried to codify what these people are doing.

In any environment there is normally a lot of tacit knowledge which helps people go about their business. Not only is this information rarely codified but it can be difficult to recognise and extract, it is often embedded in the culture and "the way we do things here.". As we delve into the process, either through writing a specification or developing code we will uncover more and more of this knowledge and much of this will lead us to change our understanding of the process.

On occasions people may choose to withhold information which we need to develop software, but often we may fail to recognise that there is information present or is relavant. Such information may be embedded in the working practises and culture of the people. For example, it may seem unimportant that every new recruit is told the story of how Old Joe managed to flood the basement one day but in fact they are being warned about the basement and the water supply.

This stuff is notoriously difficult to capture and document. Anyone who has written a pattern will recognise the difficulty in capturing just what the pattern is about and how we use it, much of the detail exists as tacit knowledge inside our heads but putting it down in a form accessible to others can be incredibly difficult.

It is inevitable that we will fail to capture important tacit knowledge when we draw up our system requirements. Successive iterations may expose more and more but some of it will only emerge when we reach testing and system deployment.

The good news is that it is easier to change systems that are rooted in tacit knowledge than those based on explicit information and agreement. Think of the rule handed down through quietly observing ones fellow employees "First one in boils the kettle". If we buy a timer for the kettle this is easy to change. However, imagine it is explicitly written into everyone's contract, agreed with unions, incorporated in the quality manual and approved by head office. Changing that is going to be a lot more difficult.

So, although it may be more difficult to develop a system when the requirements are tacit it should be easy to deploy the system. Conversely, where requirements are explicit, in say written procedures, it may actually be more difficult to integrate a new system.

Temporal dimension

Requirements documents are at best a snapshot of the way things stand at the time they are written. However things change, if we start the project on 1 January, spend a month writing documents and head back to our office to develop and test the system for the rest of the year we can be sure things will have changed in the intervening

time. Hence requirements documents need to be living documents, we may not want to accept every change that is asked for, however, setting them in stone will miss important changes.

There are few computer systems introduced today that merely automate existing practise. Instead, systems are implemented as part of an attempt to change practises. This means that to some degree the specifications are attempts to describe how things will be. Since none of us - not even management consultants - are blessed with perfect future vision it is inevitable that over time we will see changes that are needed in the proposed process and system specification.

While we have good knowledge about our internal environment and we can make plans for internal changes we have no such knowledge or control about the external environment. Things that happen outside of our problem domain can have as much, or even more, influence on what is required of a new computer system as internal events.

It is a cliché to say the pace of change in business is faster than ever before but there is at least a grain of truth in the statement. Events in the market, or action by rivals can radically change what we require from a new system. Imagine a book seller who commissioned a new stock control and retail system in the mid 1990's, they may have had the perfect specification for internal requirements but external events will have forced all sorts of changes from internet retailing to new models of revenue generation upon them.

There is a necessity for all requirements documents to be forward looking but this is also a hindrance. Again, making the document longer will make it less well understood, attempting to cover all the bases may result in a system with more bells-and-whistles than are necessary. System development cost and time may escalate and still events may over take the company.

An empirical study

A study by Edberg and Olfman (2001) looked at the motivations behind software change requests at a variety of organisation during the software maintenance phase. Corrective maintenance (i.e. bug fixing) accounted for only 10-15% of work while functional enhancements accounted for over 60% of changes. This 60% was broken down into four categories:

- External changes - changes required to meet some need from outside the organisation, say a changed legal requirement.
- Internal changes - changes required because of company changes such as new products, or restructuring.
- Technical changes - required to meet new technical demands.
- Learning - changes resulting from learning by individuals or groups.

Edberg and Olfman suggest that 40% of these changes where primarily the result of learning. By changing software organisations can pass on the benefits of one group's learning to the whole company - potentially saving money and/or time and improving efficiency.

Interestingly though, users who requested changes often didn't attribute their request to learning, they preferred to cite other internal or external factors as the motivation.

It seemed that requesting a change that would save them time, and eventually make the whole company more competitive, wasn't seen as a good enough reason to ask for a change.

Does this mean the world full of self-effacing people? No, it would seem information systems (IS) people have made their dislike of changes very clear:

> "Almost uniformly among users in work groups, there was a strong belief that the IS organization did not want to enhance software and that changes had to be justified in some way other than it would help work activities. The interviewees in IS organizations agreed, frequently commenting that the enhancements required by users were "superfluous" and, in the opinion of IS, not necessary for users to do a good job. There was a consistent conflict between work groups and IS organizations at each case about what constituted a necessary enhancement to software." (Edberg, 2001)

People learn more

While Edberg and Oldman suggest system changes are the result of learning other researches (e.g. Ang, 1997) suggests that system development can act as a catalyst for people and organisations to learn about their activities. I'd like to suggest that a natural extension of this process is for the very act of analysing and specifying a computer system will change the problem. How often does someone sit down with a manager or other office worker and enquire into what they do? How often do we attempt to map the processes that occur in our work environment? And how often does someone write a document describing what goes on?

Actions such as these are perfectly normal activities for business analysts writing a specification. However, the very act of doing them will cause people to reflect on what they are doing, why they are doing it and whether things can be done better. True, some work environments may be so oppressive that people keep these insights and ideas to themselves but other companies activity encourage people to improve their processes.

Its not only the end users who will learn and change as the system develops. The developers tasked with writing the new system will gain insights into the business and the application of technology which cause them to change their interpretation of the specification.

In fact, in coding it may not be possible to implement all the fanciful promises made by a salesman, or the vague requirements in a specification document. The coding process forces us to face the reality of what is possible and what isn't. Clients may be oversold a solution by a salesman who promises everything (at a very reasonable price), the specification may be beautifully worded to describe how these things will be brought about, but, when it comes to executable code issues can no longer be fudged.

At this point the reality of constructing a solution may force a change in the specification. These can be among the most difficult changes to bring about since such changes may not be what people want to hear about. However, this highlights the importance of keeping a feedback cycle from developers to customers and continuing a dialogue over the system requirements.

The other second system effect

Fred Brooks said:

> "The second is the most dangerous system a man ever designs. ... The general tendency is to over-design the second system, using all the ideas and frills that were cautiously side-tracked on the first one." (Brooks, 1975)

Brooks' was discussing the tendency of software developers when building systems. However, there is another *second system effect*, this time within the organisation that decides to replace an existing system which can bring about the same effects.

On the face of it, if a corporation has a working system it is to be congratulated and the story finished. But we often find companies that want to replace their existing systems. Given the reputation of IT projects to over run budgets and time one wonders why they would want to take this step but they do.

At one level, writing a second system should be easy. Get a group of developers, give them the existing system and say "Copy it." But things don't work that way. The system is usually redeveloped because it fails to satisfy some need, so the instruction is more like "Copy it and ....".

Its the "and..." bit which is difficult. The first item on this list is the immediate reason for the new system, that which the original system doesn't do. Next on the list will be all the things the original system was supposed to do but never did.

While the existing system was in place things where frozen, no matter how much people wanted things to change it wasn't going to happen. But once development on a new system begins the position is unfrozen, all that pent up frustration with the existing system can be directed as additions to the new one. Then, as people see the new one take shape the learning process is seeded and more changes will come along. However, once the new system is delivered and deployed things freeze again as the window of opportunity closes.

Resistance is...

Software engineering books are full of suggestions on how to manage changing requirements. Unfortunately many of them look at Barry Boehm's (1988) economic model of software development and note that the later changes occur in the process the more they cost, they therefore conclude that change is bad and needs to be resisted.

If we go down this route we face two serious problems. Firstly we are going to make ourselves unpopular, the software developers and managers will come to be seen the people who always say "No." Who wants a bunch of uncooperative people around the office?

Secondly, this assumes that the changes that come along after the project reaches some arbitrary cut-off point are worth less than those that came along before the cut-off point. Changes need to be assessed both in terms of the complexity they add and the value they add. Changes that come along later are more disruptive but this doesn't imply they are valueless, only that they must be worth more if they are to be worthwhile implementing.

The argument that we should resist change is based on the naive assumption that we where able to capture all the valuable requirements up-front and therefore, none that

come along later are worthwhile. However, as you can see from my arguments I don't believe this is the case.

In fact, I will go further. I think it is quite possible, indeed perhaps probable, that the most worthwhile requirements for the system will only come to light as the system develops. Only as people - both developers and clients - come to understand the new system and how they will use it will the most valuable requirements become apparent.

When we write the initial specification we document the *low handing fruit*. The specification will include the most obvious requirements, those that have been discussed before the project started, those which are already documented and those this people think of in the early stages. Yet as the project proceeds, everyone involved will get a more detailed understanding of what is happening both in the software and the company, potentially revealing even greater value in a system. Consequently, it is necessary to reprioritise our work as we go.

What can we do about this?

The software development community needs to rethink its approach to changing requirement. We need to stop seeing changing requirements as a problem and start to see them as an opportunity. If we can pin down requirements and stop them from changing then two things happen. First, our organisations cease to change - this isn't good in a dynamic business environment. Second, anyone can implement our requirements because they are fixed and known. That anyone could be a competitor company, or it could be outsource organisation with low costs.

However, when we address the changing requirements to opposite is the case. Our organisations become more flexible and can out-compete the competition because we can adapt to our environment and market more quickly. Secondly, this ability to adapt and change becomes so fundamental to the organisation that it is unthinkable to outsource it and create space between software developers and their customers.

We actively want to reach a position where new system development is generating new ideas for the business, where the specification is no longer focused on the *low hanging fruit* requirements but is addressing the most valuable.

Software development books are full of techniques to make our software development more responsive: shorter development cycles, iterative development, rapid-application development, and so on. Underpinning all of these ideas is the concept of improving the feedback cycle by making it both faster and clearer.

So, my solution to changing requirements is to improve communication between people. That is, all the people involved, the programmers, the testers, analysts and customers. And by communication I don't want to see more documents, or more e-mail, I want to see people talking to one another clearly and honestly. This means we have to value the individuals not the process or the technology.

Conclusion

Requirements change, that's a fact of life. Many IT people have adopted a mindset that change is to be resisted, indeed, many IT people have been so successful in training their customers to expect resistance to change that customers have given up. (Hardly surprising then that IT people get a bad press.)

If we look beyond the change requests themselves we see that there are good, valid reasons people request change. Potentially, through IT systems companies can get to know themselves better. Computer systems have a role to play in helping companies change.

In the current debate on *agile software development* we need to by considering the user perception of software change. What use is an agile software development if users have been indoctrinated into rigidity? For agility in software development to mean anything it must be combined with an agile organisation, we cannot view software development as an isolated activity.

Bibliography

Ang, K., Thong, J.Y. L. and Yap, C., 1997, *IT implementation through the lens of organizational learning: a case study of insuror*, International Conference on Information Systems, http://portal.acm.org/toc.cfm?id=353071&coll=portal&dl=ACM&type=proceeding.

Boehm, B., and Pappacio, P.N. (1988) Understanding and controlling software costs, *IEEE Transactions on software engineering,* 14, 1462-77.

Brooks, F. (1975) *The mythical man month: essays on software engineering,* Addison-Wesley.

Edberg, D., and Olfman, L., 2001, *Organizational Learning Through the Process of Enhancing Information Systems*, 34th Hawaii International Conference on System Sciences, IEEE, http://csdl.computer.org/comp/proceedings/hicss/2001/0981/04/09814025.pdf.

McConnell, S. (1993) *Code Complete,* Microsoft Press, Redmond, WA.