On Management: Focus, Quality, Time-boxes and Ducks

Software development is easy. I was 12 when I started programming and I picked it up no problem. Businesses are full of people who program without even knowing it: Excel is programming by another name. To paraphrase someone at the ACCU conference a few year or two ago "My organization bases its software on SOA – Spreadsheet Oriented Architecture."

Yes, programming is really easy – just read "C for Dummies." People can, and do, create small world changing programs without ever really being taught how to program. But.... Software development is also one of the hardest things human kind attempts. In fact, developing good software is so complex it might be the most complex thing man has ever attempted.

Writing a small piece of software can be (but is not always) very easy. Problems set in when you want to grow the software, want more people to use it, when you want to sell it, or package it, to re-use the code or ideas, take bugs out of the system, add features, make it run 24x7x365.

It becomes hard because: more people need more co-ordination, more people need to understand what is needed, expectations rise, objectives get confused, costs are higher, money has to come from somewhere and who ever provides it expects to get a return on their investment.

Programming might be easy but managing the effort is difficult.

Ducks in a row

It is very easy to write simple software and have it do something useful. It is an order of magnitude harder to get it to commercial quality and keep doing it. And, to make it harder, there is no single accepted method for doing this and getting it right.

It sometimes amazes me that anyone ever gets this right. More amazing is that software which is fundamentally flawed works, or at least seems to work. Stuff that by any "engineering" criteria is broken is used by companies every day and businesses depend on it.

Developing good software, delivering on time, producing with sufficient quality, etc. etc. - all the things you expect from 'professional systems' is a matter of getting your *Ducks in a Row*. When everything goes right the effort can hit the target – illustrated in Figure 1.

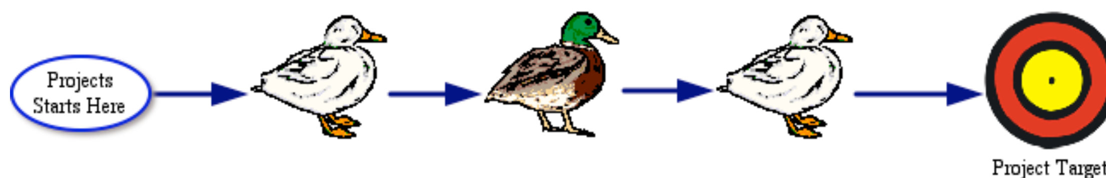

Figure 1 - A successful project

Creating good software, meeting expectations, delivering on time and the rest is not just a matter of getting one thing right. It is a matter of getting many

things "right" - or at least workable. You can get some of these wrong and still delivery something - maybe a little late, or lacking features but you will do something.

Lesson 1: On a software project there are very few things that can happen which will kill the project immediately. Failure comes through many small diversions, mistakes, errors and poor decisions. Delivering a software project is about making thousands of small decisions right rather than a few big ones.

Each time you do something badly, each time one of your ducks is out of position, you won't break the whole thing. Instead each duck out of position reduces your productivity slightly, creates a little distance from the target, or reduce quality slightly. With each duck that moves out of position it gets worse but it still works, somehow – illustrated in Figure 2.

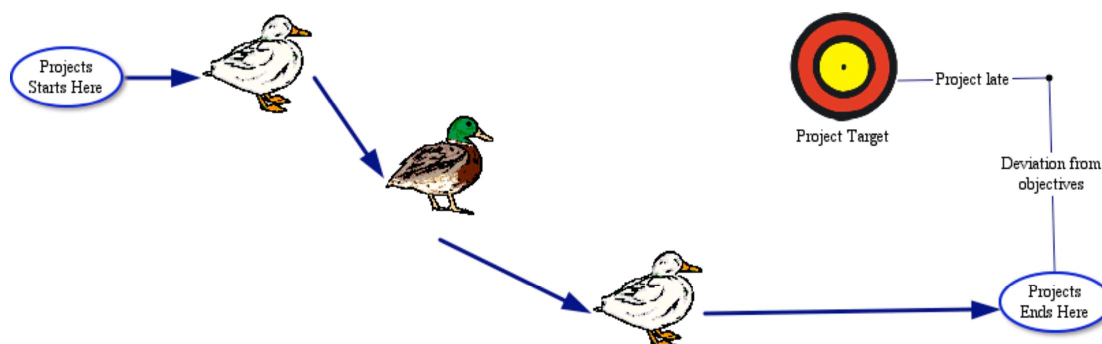

Figure 2 - Project misses the target

No one duck causes the project to fail but each duck out of position increases the distance from the target. If creating 'the best software ever' means scoring 100:

- Loose 10 points for not co-locate the development teams
- Deduct 10 points for starting work without talking about design
- Loose 10 points for spending the first three months talking about design and drawing UML
- Work to a fixed specification or work to no specification, loose 10 points.
- "We don't need no management, developers know best" – deduct 10 points
- Fire developers half way through the project, loose 10 for lost productivity, loose 10 for moral.
- Projects looking late, add more developers, loose 10 for violating Brooks Law.
- And loose 50 if you believe "Developers are the problem, our consultants say we fire them and snap together lego bricks from their toy box"

In this series of articles I hope to show how to get the ducks in a row. However every case is different so its hard to describe specifics without knowing a the specifics of a project and the problems the teams faces. All I can do is provide guidance and theories.

Quality

There are two aspects of quality: internal and external. External quality is that which other people see and use, it measures where the product is "fit for purpose." Internal quality concerns the things most users never see – in other words the software code.

External quality is obviously important, internal quality is also important but it is not so obviously. Neither quality has to be so perfect that it is flawless, it only needs to be good enough that nobody has cause to question it and ask for rework.

It is acceptable for an online purchase system to loose the odd order, say one in 1000, if the owners are accept that. Similarly, it is acceptable for software code to use procedural programming where object-oriented might be better provided the code works and is maintainable.

Quality doesn't mean gold-plating systems or over engineering systems. To do so adds to cost without adding significantly to benefits. Provided quality is high enough not to cause future problems then it is good enough.

When quality falls below this level problems emerge. In the case of software this means bugs and poor usability. When this happens there are two effects.

The immediate, or direct effect is the problem itself. A customer finds a bug, the customer is not happy, the bug has to be reported, fixed, tested and the software reshipped. All this takes time and money.

The second, indirect effect is perhaps more damaging still: the workflow is disrupted. Rather than doing what they were supposed to be doing managers, developers and testers have to devote time to rework. Either the usual routine is disrupted or extra resources must be kept ready to fix problems.

Lesson 2: Rework costs far more than most people realise. Many of the costs are hidden.

One of the reoccurring problems in the software industry is the willingness to pursue short-term objectives – such as meeting a deadline – rather than invest in quality. The competitive environment start-up companies find themselves in sometimes allows no other choice. But accepting lower quality today is a false trade off, it stores problems for tomorrow.

Lesson 3: Accepting lower quality today stores up problems for later.

The software industry is not the first to face this problem. Nearly 30 years ago Philip Crosby identified and described these problems in other industries. This lead to his seminal book *Quality is Free* (Crosby 1980). This lesson has been described again and again in industries such as car manufacturing (Womack et al. 1991) and semi-conductor manufacturing (Reid 1985).

Those managing software development projects need to put a greater emphasis on quality - both internal and external – than is often the case today. The days when making a deadline meant trading quality for time are over.

Lesson 4: The software industry needs to collective raise the standard of produce and reduce tolerance for rework.

Raising the game on quality starts with attitude: deciding higher quality is essential. Then it moves to approach: allowing time for quality, rewarding quality work and prioritising quality over quality over new features. Approach gives way to practices: requirement inspection, code reviews, test driven development and other techniques.

Critically quality cannot be tested into a product. Improving quality is not about employing more Software Testers and fixing more bugs. It is about stopping bugs from happening in the first place. Again there isn't anything new here, W. Edward Deming preached this mantra from the 1940's onwards.

When faults do occur they are the result of the production system not the individuals performing the work. Systems allow, even encourage, individuals to make mistakes. If a developer creates a bug it is not because the system they are working within allowed the bug to be created.

Lesson 5: Systems, not individuals, are responsible for bugs. Improve the software production system to improve quality.

The reward for improved quality is not just happier customers, it is less disruption in the workplace, improved product flow, happier workers and more predictable results. Or, as Philip Crosby would say: *Quality is Free*.

Rework is lost work

The cost of work to fix faults – and bugs – is graphically demonstrated by the contrast between General Motors (GM) and Toyota told in *The Machine that Changed the World* (Womack et al. 1991).

The researchers visited a GM car factoring in 1986 where they found the production line running at full capacity constantly. Anything that might stop the production line was to be avoided because that would mean lost production. When the car reached the end of the line it was inspected for faults that may have occurred during production.

Cars that failed inspection were put in a queue for repair. The direct costs of repairs included: the parts to replace, workers time to do the fix, storage space while the car was waiting for repair and then re-inspection (testing). There were also indirect costs because the company could not predict when a particular car would be ready to ship. The repair process did not disrupt assembly directly because it was so routine but it did require management time to administer the process and extra (specialist) workers.

Next the researches visited a Toyota factory. Here all workers looked for defects as the car was being assembled. If one was detected the whole production line was stopped, the fault was fixed and an investigation launched into how the fault had occurred. When the reason was found the cause would be fixed and the production line restarted.

At first sight Toyota might not have the throughput as GM but the savings from the repair work more than made up for this. Over time Toyota debugged their production

system – rather than faulty cars – their production system actually got faster and faster. Productivity was higher than GM.

Time boxing

It is traditional software projects to ask individuals how long they thing each task will take, aggregate the estimates – perhaps using a Gannt or PERT chart – and produce an estimated completion date. Almost anyone who has ever worked on a software project with formal project management will recognise the technique. But, estimates are just that: estimates. In the world of software engineering estimates are notoriously poor indicators of how long a piece of work will take,

There are exceptions, some individuals and organizations monitor their estimates, apply adjustment factors and as a result have reliable estimates. However these organizations are the exceptions, most struggle with estimates. For such organizations estimates are more trouble than benefit.

An alternative approach is to turn the question around. Fix the length of time available and ask *What can be achieved in this time?* Reduce each work item into small pieces which can deliver a benefits and can fit within the time allowed.

This approach is called time boxing. People find it difficult to accurately estimate how long something will take so instead they are asked what they can fit in a given time. Then they are asked to commit doing the work in the time given.

Importantly time boxes are the same length and are relatively short – a one or two weeks is typical. Because work is divided into a series of time boxes of the same length people get fixed reference points to measure their progress by.

Lesson 6: People are poor at estimating how long a piece of work will take. So turn the problem around. Set the time allowed and ask: *What can be achieved in this time?*

One time box is followed by another. When a piece of work is large it is broken down into several smaller pieces and allocated over several time boxes. If there is any question of the work not being completed within the time box then it is broken down into several smaller pieces. Each piece of work should represent some measurable improvement in the system.

Using time boxes creates a rhythm to work. People are better able to understand what can be achieved in any set period when they always work to the same schedules. Like the rowers in a boat race, there is a predictability that allows co-ordination.

However time boxing does not answer that age old question: *When will it be ready?* If experience teaches anything in software development it is that this question cannot be answered with predictability.

Lesson 7: *When will it be ready* is not a useful question, and the answer is almost certainly wrong. Such questions and answers lay the foundations for disappoints.

Again the question needs to be turned on its head. This time the one question is replaced with two: *When do you need it by?* And *What is it?*

When we do this we change the nature of the conversation. Rather than an open ended discussion between two or more parties about when some item may be available the conversation becomes one of prioritisation and possibilities. Confrontation is turned into a negotiation.

These two questions allow the discussion to progress by focusing on what the customer requires and when. *When will it be ready?* is not a useful question for a software developer, it tells them nothing of priorities and needs. Knowing exactly what is needed and by when is much more useful.

If the *what* cannot be delivered by the *when,* then conversation can proceed to break down the what into smaller pieces and possible compromises or interim solutions. In contrast, if the answer to *When will it be ready?* is not acceptable to the questionnaire the conversation tends to become confrontational.

Lesson 8: Schedule discussions need to be rich conversations. Understanding what the customers require and when they require it allows alternatives to be discussed and options examined.

Shared Focus

Leave to one side the distractions of the work environment, office politics, the office move, the state of the kitchen. Leave aside too personal distractions, painting the house, cleaning the car, your new girlfriend. Leave aside the distractions of everyday human life, this morning's late train the football match and the new sandwich shop. Software projects need clear focus.

Software development is no longer an individual sport; large programs and systems are created and operated by teams. All members of the team need to be focused on the same thing and pulling in the same direction.

Unfortunately software projects are full of distractions. Ambiguous requirements may give different team members different understandings of what is required. Time schedules that lack credibility mean nobody knows when a product should complete let alone when it will. Debates over technical merits of technology X over technology Y mean neither is used.

Successful teams need focus. Focus removes distractions and channels all efforts in one direction. Individuals need to share objectives and work together.

In order to hit a target the team needs to aim at the target. So the team needs to know as much about the target as possible: what is it, when is it, and why is it and where is the team now. Sports teams always know the score, software teams need to know where they are relative to the target.

Lesson 9: The first responsibility of any leader is to create a shared focus. Squeezing out ambiguity, wherever it is found, allows work to proceed more productively.

The larger the work effort and the longer the time scales the more difficult it is for a team to hold shared focus. In order to create focus software development teams need to shorten their time horizons and reduce the number of things they are attempting to accomplish.

Time boxing is one technique that can help here, work is reduced into small pieces which can be forced on for a short while. Raising the quality bar is another, when everyone knows quality will not be compromised then the discussion goes away.

In developing software there are many, many, options and consequently many decisions to be made. After all, software is soft, everything is malleable. The development process is one of turning ideas into physical code, from abstraction to execution.

The temptation is to keep all options open and allow infinite flexibility in the development process. However this only increases the decisions to be made and provides more opportunities to make inconsistent decisions – to move a duck out of position.

Closing options, denying ourselves some of the tools helps reduce the decision space and the unpredictability encountered. Setting out a framework in which to operate reduces uncertainty and increases focus.

Summary

If teams are to be focused they need to know what they are trying to achieve. If work is broken down into meaningful pieces somebody needs to know which pieces are meaningful and which are waste. When work is worth doing it is worth doing well, with high quality. Doing unnecessary work poorly is not a compromise but a waste.

Project management is about delivering something, in some time frame. Project Managers are trained to report to manage that delivery, to report on progress and to adjust the delivery process to meet the time frame.

Project management is the dominant paradigm in the software development world. As such it tends to squeeze out the other aspects of management but it is not the only aspect of management in software development. Getting the ducks in a row involves other aspects of management.

Project success however should not be defined by delivering something but on delivering meaningful value to customers. Work packages should not be sliced and diced to fit a schedule, they should be sliced and diced to deliver value. The *what* is delivered is key to determining the *when*.

Future pieces in this series will look at other management aspects of software development and specifically at managing the *what will be delivered*.

References

Crosby, P. B. 1980. Quality is free : the art of making quality certain: New American Library.

Reid, T.R. 1985. Microchip Glasgow: William Collins & Sons.

Womack, J.P., D.T. Jones and D. Roos. 1991. The machine that changed the world. New York: HaperCollins.