On Management

As usual Fred Brooks got here first:

> "In many ways, managing a large computer programming project is like managing any other large undertaking - in more ways than most programmers believe. But in many other ways it is different - in more ways than most professional managers expect." (Brooks 1975)

A few years later he pointed out how important management is:

> "Some readers have found it curious that *The Mythical Man Month* devotes most of the essays to the managerial aspects of software engineering, rather than the many technical issues. This bias ... sprang from [my] conviction that the quality of the people on a project, and their organization and management, are much more important factors in the success than are the tools they use or the technical approaches they take." (Brooks 1995)

Managing software development is a big topic. It is a mistake to equate the management of software development efforts with *project management*. There are project management aspects to the topic but they are a subset of the whole. Indeed, the discipline of project management openly acknowledges this. For example, the UK Government backed PRINCE 2 project management techniques excludes all human resources aspects of management.

PRINCE 2 defines a project as:

> "A temporary organisation that is needed to produce a unique and predefined outcome or result at a prespecified time using predetermined resources." (Commerce 2005, p.7)

While I'm sure this describes the situation some readers find themselves in, I'm also sure that many many more of you find yourselves in a different type of organization. You are working on something that doesn't have an end date, or if it does there will be another "project" starting on the same code base the next day.

Rather than call these efforts *projects* a better term is *products*. Products unlike Projects go on and on so I prefer the term *product*. This introduces a longer time perspective and emphasises the need to produce something tangible from the work.

Product Management is a discipline in its own right. One that is understood much better in Silicon Valley and the US than it is in the UK and Europe. You can replace the word Project with the word Product but you can't replace a Project Manager with a Product Manager as the roles are different. More importantly the skills needed for each, and the training given to each, are different.

Then there is all the other management stuff: recruitment, retention, assessment, business strategy etc. etc. In other words: there is a lot to be said about management in the software development arena.

Unfortunately a lot of people have come to believe that "project management" is the way to manage all IT. It isn't. There is a lot more to "software development management" than managing the project. Limiting our view of management to "project management" risks harming our work.

So I have decided Overload needs a new series, *On Management.* We'll start with Project Management, move through into Product Management and take in some of the other stuff along the way. No time scales, no promises, no defined route, design will be emergent.

In this, and future, articles, I will not hide my agreement with Agile and Lean thinking. Indeed I will take many of the Agile practices as given. Agile is a brand, a powerful brand, and a brand that gets most things right. But it is also a brand that gets peoples backs up. It's also a brand that doesn't go far enough in some respects.

When it comes to management most Agile management practices are just plain *good management*. I know not everyone agree with Agile ideas – and I don't agree with every word ever written about Agile development – but at present I think Agile represents the current state of the art.

Product management, strategy, IT strategy, financing, human resources – recruitment, retention, objective setting, compensation, succession planning, and more – and more. There is plenty of material here. So best to get started...

Triangle of constraints

All software is developed under constraints but there are three which are more important than others: time, resources and features (McCarthy 1995).

Others could be added, money being the obvious: Money is, economists like to tell us, fungible. Which is another way of saying it can be exchanged for other things very easily. Money can be exchanged for resources such as a new developer, thereby increasing our resources. Or money may pay for overtime working thereby increasing the time we have on a project.

The net result is that introducing money complicates things. Since (almost) everything can be reduced, or replaced, by money this analysis leaves money to one side. Rather it is better to regard cost as a function of time and resources, and revenue as a function of features. If we increase the time or resources then costs will increase, and if cost needs to be reduced then resources and/or time needs to be reduced.

Resources is a rather elastic word as well and can include just about anything. In the name of simplicity, in this context resources is taken to mean: people (developer, testers, etc.) and the tools they need to do their job.

These three parameters can be thought of as a triangle:

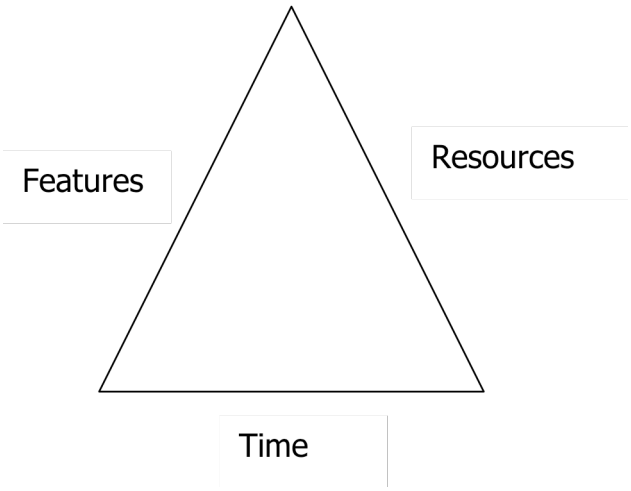

Figure 1 - Triangle of constraints

Lesson 1: Time, resources and features are the critical factors that require managing. But they are not the only factors.

All software development takes place within such a triangle. As with any triangle it is not possible to change one of the three parameters without changing another:

- More features must be accommodated by either increasing the amount of time available or adding more resources.
- Delivering a project in less time requires more resources or a reduction in features.
- Adding more people (resources) to the project either increases the amount of time it will take or, in theory, allows for more features – except...

The People issue

That last bullet point sounds OK, doesn't it? Except the way it usually works is that adding more people invokes Brooks' Law:

> "adding more people to a project a late software project makes it later" (Brooks 1975)

Adding people to a project comes at a cost. New people need time to come up to speed on the system being developed, the requirements, the existing code base, the technology, etc. etc. Consequently, in the short run the resources on a project are effectively fixed.

In the long run people can be added to a project, and they can increase the capacity to undertake work but they come at a cost Therefore, as Brooks' Law states, if the project is late adding more people will make it later.

However, if a project is not late, or rather if the project is managed actively, people may be added to the project, without too much detriment. Projects which plan to add people can do it in an orderly fashion.

Lesson 2: Adding people to a project needs to be done in an orderly fashion.

In fact, it is essential to add people to a project over time because there is a natural tendency for people to leave a project. People get offered better jobs, people take time off for health and personal reasons, overseas workers decide to go home, and people retire.

Lesson 3: Active management seeks to slowly expand a team to compensate for natural loss.

Obviously there are times when this is inappropriate, such as when a project is winding down. There are also occasions were it is more important to add people.

The net result of these forces is that, for any project, in the short term the resources available are fixed or even reducing. (The short term may be as short as three months or as long as a year.) Only in the long term can resources be increased and even then major increases in resources are not possible.

Consequently, managing software development becomes an exercise in:

- Human resource management: motivating people, retaining people, hiring people and training people.
- Managing the time v. feature trade off.

Neither of these trade-offs is, strictly speaking, a Project Management task. Project management techniques like PRINCE 2 explicitly exclude managing people. While a Project Manager may be able to offer advice on time considerations, the decision on whether to include or sacrifice a feature is a job for someone well versed in business need. This is a job for a Product Manager or a Business Analyst.

Lesson 4: Human Resource Management is not part of Project Management. However, when managing a project many of the issues are people issues.

Time v. Features

It's obvious really: the more time a team has the more work it can do and the more features it can implement. However, the longer a piece of work is scheduled to last the greater the expectations and the greater the risk.

The future is uncertain, the degree of uncertainty increases in proportion to the length of time considered. Next week is more uncertain than tomorrow, and next year more so. A competitor may launch a product and steal the market, legal changes may limit the products application – as happened to some online gaming companies – or economic changes may render the software unprofitable.

Neither is it just risk that increases with time, technology advances. New operating systems, new chips, new discoveries may undermine the software under development or require re-work.

Lesson 5: The further you look ahead the greater the uncertainty.

In order to cope with these difficulties – and others – it is necessary to consider shorter time frames. There is significantly less risk attached to product development which lasts six months than one lasting two years.

Less risk equates to less cost but there is also revenue to consider. A product that ships in six months will start earning revenue for the builder in a quarter of the time it takes the longer project. This means cash will start flowing that much sooner – especially useful for start-up companies.

However, shipping a product in a reduced time frame creates two problems, one technical and one social.

A technical problem

Technically software engineers are taught to, well, engineer. To design systems that are resilient to change and will stand the test of time. To stand like a bridge for a hundred years. But software faces different economics to bridges and buildings.

Unlike most construction projects most of the cost of software occur after it is initially released – what is euphemistically called the *maintenance phase*. It is hard to foresee the changes that are required during this phase.

A building may be designed by one individual, or by a small group of individuals. It is then constructed by another, larger, group of people. However, there is little design, innovation or problem solving during this phase. Much of the work is performed to industry standards. Therefore the final structure mostly resembles the original design.

Design, innovation and problem solving occur at every step of software development. Deciding whether to divide a piece of work into several classes each with one function or, one class with several functions is a design decision left to individual developers. The scale of the task is such that the designer, or architect, cannot have sight of all the decisions unless they actually perform the work themselves.

Consequently software is the ongoing work of many minds rather than a few. Naturally there will be differences of opinion and approach.

Software development is often opportune, if released at the right time the software can fill a market need and make profit. Releasing the same software later may miss the opportunity. Therefore the pressure to "get something" delivered is high.

A late product, no matter how well engineered it may be is often worthless. But a timely product, no matter how bad may be worth millions. This dilemma creates the conditions for adverse selection. Poorly engineered or designed products may often be better positioned to win. This has problem called *worse is better* (Gabriel 1990).

These problems bedevil software developers. Software engineers have yet to find ways of developing software that allow for *good design* without imposing excessive economic costs. Test-driven design, rough up-front design and refactoring are part of that solution but not the entire solution.

The maintenance phase corollary

Most people who have formally studied software development and engineering will have been taught that 80% of the cost and effort expended on software occurs not in the *development phase* but rather during the later life time of the software, the *maintenance phase*.

But far fewer people appreciate the corollary of this. If this rule holds for all software it follows that 80% of a developer's career will be spent maintaining existing software, or possibly that 80% of developers will spend their entire career maintaining software.

Given that it might be reasonable to assume that 80% of the research into software development considers the maintenance phase, or that 80% of the publication relate to maintaining software. Yet neither seems to be the case.

Prioritisation

The second problem a reduced timeframe creates is the need to decide which features are included and which are left out. According

to our triangle, with fixed resources, if we reduce time we must reduce the feature set.

Unfortunately this requires tough choices. Development projects are often like trains. They don't leave the station very often and when they do you are either on it or you are not. People will pay a lot of money to be on a train, or squeeze themselves into a small space rather than wait for the next one. Worse still, with software projects it is not always clear that there will be another one.

Consequently lots of people want their requests included in a software project. Since including a request is relatively cheap there is little incentive not to include it. Indeed, not including a request risks offending or upsetting someone, therefore there is an understandable momentum for including it.

At some point decisions about which features are included and which are not need to be made. Postponing these decisions is bad for the development team because they have to consider all requests – or at least read the documents – and most likely spend time discussing requirements with stakeholders.

Postponing decisions makes sense not only from a social point of view but also from a business point of view. The option to develop a new feature, or not to develop a feature, is exactly that, an option. Economics, again, shows that options are valuable. (If you want to know the details read up on Real Options which apply ideas from financial options to real life problems.)

What is needed is a clear prioritisation process for the development team. The team need to know what work is required for the next development period and which is not. They should then ignore all other requests, to consider any element would unbalance the economics.

In order to have clear prioritisation somebody – or some group of people – must be able to make a decision. This individual needs to have all the information necessary to make the decision, they must be trusted by the organization and they must be empowered to make these decisions and make them at the right time.

This role is that of Product Manager. Not all organizations have Product Managers in name, they have different titles, like Business Analyst or Product Owner, but many organizations simply do not have Product Managers at all.

Lesson 6: Product Managers are needed to decide what goes in, and what does not go in, each software release.

In some organizations Project Managers fill this function. The problem here is that Project Managers are trained in a different skill set. They are trained for estimating, project scheduling, risk

assessment, issue and progress tracking, reporting and such. They are not trained to gather information from disparate sources and make business value judgements.

Priorities should be communicated to development teams in unambiguous terms. The simplest way to do this is to prioritise requests as 1, 2, 3, and so on where no two items are allowed to have the same priority. So there is only one number one priority, one number two and so on.

Lesson 7: Priorities need to be unambiguously spelt out to teams

Some organizations use the so called "MoSCoW" rules to categorise items as Must Have, Should Have, Could Have and Will Not Have (or Would like to Have). Such prioritisations are an abdication of responsibility on the part of the business. Asking a team to develop five Must have features turns over the decision to the development team, when this happens the business loses its right to complain about the result.

Conclusion

The triangle of constraints governs all software development. Add to it Brooks' Law and all decisions come down to questions of how long a project will take, and which features are included.

To date software engineering has done developers a disservice by allowing engineering to become top heavy. New engineering techniques are needed that can be used in short cycles.

The business side of work also faces a challenge: to straighten out the prioritisation process. There is one ready made answer: to embrace Product Management but unfortunately too few organization are using these techniques. Neither is this any guarantee, product management can be done badly or it can be done well.

And this is just the tip of the iceberg when it comes to managing software development. A future article will discuss the role of Product Management in depth, but before then, the next instalment will discuss quality, time-boxing and focus.

Acknowledgements

Thanks to Ric Parkin and the Overload team for comments and suggestions.

About the author

Allan has experience both as a software developer and a development manager – largely with independent software vendors whose very survival depends on their ability to ship software. He is a regular conference speaker and contributor to publications on the subject of

Agile development and improving software development. His first book, "Changing Software Development: Learning to be Agile" was published by John Wiley & Sons in 2008.

He is a trained product manager and project manager – holding PRINCE2 Practitioner status – in addition to his a BSc in Computing and Financial MBA.

Allan currently works as a consultant and trainer helping companies organise their software development activities. He can be contacted at allan@allankelly.net.

More pieces on this topic and others can be found on his website: http://www.allankelly.net.

References

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

Brooks, F. 1995. The mythical man month: essays on software engineering. Anniversary edition Edition: Addison-Wesley.

Commerce, Office of Government. 2005. Managing Successful Projects with PRINCE2. Fourth Edition. London: TSO (The Stationary Office).

Gabriel, R.P. 1990. "Worse is Better." In EuroPAL. Cambridge.

McCarthy, J. 1995. Dynamics of Software Development: Microsoft Press.