The scoping problem

It is a cliché of software development that every project is understaffed with a deadline that is too tight, and maybe under resourced in terms of hardware and software too. I mean, have you ever worked on a project where there where too many programmers?

Jim McCarthy (1995) suggests that every project is bounded by three factors: features, time and resources (i.e. developers) – these form the scope of the project: what do we need to do? How long have we got? who's doing it? The product of these factors is a constant: if you want more features you must either increase the time allowed or the number of developers. Likewise, if you want a delivery sooner you must either reduce the number of features or increase the number of developers. But, as we know from Brooke' law (Brookes, 1974), "adding manpower to a late software project makes the project later." Unfortunately Brook's law is not true in reverse: reducing the number of developers will also make the project later. Brook's law can only be applied to a given point in time.

McCarthy's three factors neglect quality. If this is a fixed value McCarthy's theory holds. Alternatively, we could attempt to reformulate the equation as:

Time x Resources = Features x Quality.

Although this equation has a nice symmetry to it seems break Brooks' law because it implies that people can be traded against time. In the short run this is certainly true, remember Brooks' law applies to "late projects", in the longer run we may be able to usefully add more people to a project. Clearly there is no simple equation for software development.

I'd like to add two observations of my own to McCarthy and Brooks:

Observation 1: Developers want to exceed the project requirements. Sometimes this means they want to see a feature which isn't in the spec, sometimes it means they want to do frivolous things like refactor the code, or make it more readable, heaven forbid, some of them actually want to make the code re-usable! Truth is, most developers believe they know best. Some of us are even known to claim that this is a professional asset.

Observation 2: Developers are more likely to underestimate the time taken to do a piece of work than they are to overestimate; this is especially true when a developer wants to do a piece of work.

From these observations flows the axiom: software projects are naturally over committed in terms of time and resources.

If my observations have not convinced you consider what would happen if you turned around at your next project meeting and said: "Well than chaps, looks like we've got four more weeks work and then we'll be finished a whole month early, well done." Would the reaction be:

a) Launch a month early: pull the deadline back!
b) Add in some more features, fix some more bugs: back to work!
c) Fire the contractors we can save some money and still get it done on time
d) Congratulations: take a months paid leave

I'd like to be bold and suggest *Kelly's first law of project complexity*:

Project scope will always increase in proportion to resources

You see, I think projects are bit like programs: they will expand to fill all the time and human resources available. This may be a generalised case of Brook's law, in his case he adds more developers, now the project scope increases to cover training new developers too.

Every time a deadline is pushed back to allow more bugs to be fixed you increase the scope too. Even if the project introduced those bugs in the first place fixing them was never on the feature list.

With a slack deadline there is a tendency to try some new beta OS, upgrade your compiler, or experiment with generic programming or many other things. Yes, these all have a place I'm just saying you have to recognise the cost of all these things. Introduce a breathing space between projects to do these things.

I don't want to argue for arbitrary deadlines either. A deadline picked from the sky and imposed is just as bad. Most projects have a rough deadline before they ever start, given this the team needs to look at the feature set, the resources and balance all these things.

A word of caution: a developer joining a project at mid-point may not be aware of these discussions, it is important to explain the schedule to them and even adapt it after hearing their views.

If all of this puts you in mind of short cycles you're right. Every time I work on a large project I am convinced that much of the project could be achieved with less code, complexity and effort. This leads me to *Kelly's second law of project complexity*:

Inside every large project there is a small one struggling to get out

Take a look at your project, why is it so big? Would you write it the same if you were doing it again? Chances are you would not, you have learned from what you have done, and you would re-write it using less complexity and less code.

Writing a program is teaching the machine how to do something. In doing this you come to understand the problem intimately and inevitably see better ways of doing it. So much complexity is unavoidable, but I suggest that much of it is avoidable if you merely reduce the scope of the project.

OK, how do we reduce the scope of the project? Well, firstly aim for minimalism (Henney, 2001) in design. Secondly, design extendable systems. Write a core system to which you can add code, your project may end up being big but it will still contain a micro-kernel like element which will make it easier to get to grips with.

Next: keep your project and your team focused. Each project iteration should have some *Big Idea* which to can drive towards. This is good for morale too and you always know what when you have arrived.

And try this on your management: for every day the project is finished early, everyone on the team gets half a day's paid holiday.

Bibliography

Brookes, F., 1974; The Mythical Man-Month, Addison-Wesley, 1974

Henney, K., 2001; Minimalism: omit needless code, Overload 45, October 2001

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