NUWAY

The Keys to Software Projects

Four simple rules for success.

FEBRUARY 2014

Software projects are renowned for delivering late, over budget, and sometimes even not at all. Even successful projects produce software that is hard to use and poorly matched to user's needs with distressing frequency. As software influences more and more of life, the Holy Grail of effective and efficient software projects becomes increasingly important.

Ultimately software projects fail because they neglect to follow the principles that govern the software lifecycle. Like the natural laws discovered by scientists in the 18th century that govern the physical world, these principles are not a statement of how things should be in software projects. They describe how things are, and we ignore them to our peril.

THE TWO-PIZZA RULE

The first law of software projects is that small teams are significantly more productive than large teams. One study showed that smaller teams[1] – defined as two people on small projects or three people on large projects – were 39% more productive than large teams. These results were validated by the QSM study, which examined over a thousand software projects conducted between 2005 and 2011. The study revealed that on small projects, big teams (averaging about 8.5 developers) completed the project 24% faster than small teams (averaging 2.1 developers) – a bit more than a month over the course of a four and a half month project – but at three times the cost. For larger projects, however, the difference is even more dramatic: big teams complete the project only 6% faster (a mere 12 days over a ten month project), but at four times the cost.[2]

This research proved what savvy software leaders already knew. The CTO at Amazon, Werner Vogels, coined the "two-pizza rule": if a project team can eat more than two pizzas, it's too large.[3] Observations from Google and Nokia have also demonstrated that teams with an upper limit of four or five people are more productive. The QSM study backs up this anecdotal data, showing that the best performing projects used 4-person teams, while the worst-performing projects had 17.[4] Teams of three to five people performed well across the board.[5]

Why does this happen? The QSM study points to a single factor that explains the high productivity of small teams: fewer defects. Larger teams had double the number of defects on small projects and three times the defects on larger projects.[6] The time wasted finding and fixing these defects (and finding and fixing the defects that these fixes introduce, etc., etc.) is why larger teams are outperformed by smaller teams.

But why do smaller teams have such a lower defect rate? No conclusive research has identified the cause, but two factors have been offered by way of explanation. First, smaller teams have a lower communications overhead than larger teams. In 1975, Fredrick Brooks observed that "adding manpower to a late software project makes it later."[7] This is due in part to the initial effort required to orient the new person to the project, but far more effort is lost to the increased communication overhead required to keep the entire team aligned – or to the consequences of not maintaining that overhead.

Secondly, as teams grow, individual performance is lost to group dynamics. Not simply are the superior efforts of star developers diluted in a sea of mediocrity, but their abilities can actually be blunted in the absence of a culture of excellence. This dysfunctional group behavior is known in sociology as the Ringelmann effect.

The phenomena described by Brook's Law and the Ringleman Effect prompted Walt Scacchi to observe that the use of small groups of 3-7 experienced developers was a significant contributor to high software productivity.[8] This brings us to the next principle of software projects: use the best developers you can find.

THE EXPERT MULTIPLE

One law of software projects that has enjoyed repeated validation over the span of decades is that the best software developers are ten times better than the worst. In 1968, Sackman, Erickson and Grant observed that the best programmers were 28 times better than the worst.[9] Later studies have confirmed that disparity, although many propose the more modest ratio of ten to one.[10] This ratio holds regardless of the metric – productivity, defect rate, code performance, etc. Even when compared with average developers (rather than the worst), the best are still 2.5 times better.[11] This ratio also holds when controlled for experience – that is, it holds amongst people with the same level of experience, and even between people with differing levels of experience (i.e. the best intermediate people can be 2.5x better than the average senior people).

A number of explanations for this performance gap have been offered. One that enjoys the most empirical proof – probably because it is the easiest to measure – is that the best programmers write code with fewer defects. One classic study examined how programmers with at least four years of experience debugged a program with twelve defects. The best developer found all the defects and didn't introduce any new defects while correcting them, while the worst missed 4 defects and inserted 11 new defects while fixing the 8 he did find.[12] McConnell extrapolated this trend to discover what would happen over successive debug cycles (i.e. how many cycles it would take to remove all the bugs, including the bugs introduced while fixing previous bugs). The top three developers would take three debugging cycles, while the slowest three developers would need 14 cycles – almost five times more cycles. When you take into account that the slowest developers take three times longer to execute a debugging cycle, this means that they need 13 times longer to fully debug their programs as the top developers.[13]

In addition to quality, Haack suggests three less measurable reasons why expert developers are so effective. First, they are able and willing to take initiative: "If they get overly stuck on a problem, they'll come to you or their coworkers and resolve the problem." They also write code that is easy to understand: "By writing maintainable code, a good developer can make changes more quickly and also improves the productivity of his or her team members who later have to work on such code." Finally, they also write less code. This means they need less code to accomplish a given task – the Sackman study found that the best developers' programs were 20% the size of those written by the worst. But they also know when not to write code, instead using code written (and tested) by others, such as third-party libraries.

These factors, together with high quality, show why the use of "experienced development staff" was found to be a software project attribute that facilitates high productivity.[14]

THE QUALITY SHORTCUT

In 1979, Philip Crosby published a book with the deliberately provocative title, *Quality is Free*.[15] While this has been misunderstood to mean that quality is easy, the original sentiment is that *quality pays for itself*. To put it another way: the effort spent on upstream quality improvement and assurance tasks is more than offset by the reduction in the high costs of resolving defects downstream. This what Crosby meant when he said, "Quality is free. It's not a gift, but it is free."

Several studies have shown that defects are easiest to resolve close to the point at which they are created. Contrarily, the more time that passes between the point at which a defect is introduced and the point at which it is detected, the more expensive it is to fix. McConnell examined the findings of eight studies conducted between 1974 and 2004 and summarized them in the table on the following page.

As the table shows, a requirement defect (i.e. capturing a requirement that turns out not to be correct) is easy to resolve while the project is still in the requirements stage, but requires three times more effort to fix in the design stage (due to numerous design decisions having already been made based on that faulty requirement). Even worse, RELATIVE COST OF FIXING DEFECTS BASED ON WHEN THEY'RE INTRODUCED AND DETECTED[16]

	Phase detected				
Phase introduced	REQUIREMENTS	DESIGN	CONSTRUCTION	SYSTEM TEST	POST-RELEASE
Requirements	1	3	5-10	10	10-100
Design	-	1	10	15	25-100
Construction	-	-	1	10	10-25

if the requirement is only discovered to be incorrect while the project is in the construction stage (i.e. actually writing the code), it takes five to ten times more effort to resolve. Worst of all is if the defective requirement is only identified after the project has been released; in that case, it requires up to one hundred times more effort to fix than if it had been caught right away. Similar dynamics are at work with design and construction defects.

The key to early and effective defect detection is to employ a broad range of quality improvement and assurance techniques throughout the life of the software project. Certain defect removal techniques have been shown to be more effective than others. The top four techniques are: large-scale beta testing, including more than 1000 sites (75% defect removal rate); prototyping (65%); formal code inspections (60%); and formal design inspections (55%).[17] Sadly, the most commonly used means of defect detection – unit testing, integration testing and regression testing – only have a detection rate of 25% to 35%. However, even more important than using the best technique is to use a variety of techniques, because different methods find different kinds of defects. One study found that using two different techniques doubled the number of defects found.[18] Moreover, the study found that two different groups using the same technique also doubled defect discovery. Therefore, a successful software project employs a diverse range of people and techniques in order to discover as many defects as early as possible.

ITERATIVE AND INCREMENTAL DEVELOPMENT

Specifying a software system is hard. Very, very few people have the capacity to envision how a system should work entirely in the abstract. Most benefit from actually seeing something concrete – an early release, a prototype, even a screen mockup or wireframe drawing. Only when they see an implementation of the proposed system can they confirm that's what is needed, or in what ways it needs to change. Unfortunately, the presiding paradigm of software project management is to capture the system requirements in a specification document. Since few people can really tell if the software described in a document is what they really need, there are often unpleasant surprises at the end of the project when they get to see the software for the first time.

The solution is to adopt an empirical approach instead of a speculative approach. In other words, instead of trying to guess what is needed ahead of time, build a little something and then evaluate how it needs to change to better fit the situation. One common empirical practice is to develop a prototype. A survey of 24 case studies revealed that prototyping leads to better designs, better matches with user needs, and improved maintainability than in specification-based development.[19]

However, prototyping by itself is insufficient. The chance that the team developed exactly what was needed in the first prototype is extremely unlikely. Even if they had, a prototype is not a production-ready system. Therefore, a completely empirical approach would be to release multiple prototypes, checking the project team's work at frequent intervals with the stakeholders to endure that they are on the right track. This approach is both incremental (the system is built in progressive stages) and iterative (increments are repeated until the system is finalized). The advantage is that at each stage, the team focuses on creating the simplest possible system that meets the actual business needs. This results in avoiding creating unnecessary functionality – a situation that often arises when creating a speculative requirements document at the beginning of a project. As Maxwell observes, "Eliminating extra features that no one really wants represents the single largest opportunity for increasing software development productivity in most organizations."

CONCLUSION

A software project that takes these laws into account would look different than many currently being executed. It would use a small team, or clusters of small teams working on different parts of the overall system. The team would be composed of the very best people that could be found, and they would prioritize quality over all other non-functional requirements. Finally, the team would work on the project in an iterative and incremental fashion, growing the system over time and presenting the fruit of their work frequently to the end users. Projects conducted in this fashion will produce great systems efficiently because, instead of resisting the laws of software, they will be working with them.

Tom Sweeney, BEng, MSc
tsweeney@nuwaysoftware.com
is the Manager of Innovation with Nuway Software, based in Burlington, ON, Canada.
www.nuwaysoftware.com

[1] Boehm, Barry W. , T. E. Gray, and T. Seewaldt. 1984. "Prototyping Versus Specifying: A Multiproject Experiment." IEEE Transactions on Software Engineering SE-10, no. 3 (3): 290–303.

[2] http://www.qsm.com/blog/2012/part-ii-small-teams-deliver-lower-cost-higher-quality

[3] http://www.agiledevelopment.org/agile-talk/for-higher-productivity-should-go-even-smaller

[4] http://www.qsm.com/blog/2012/part-ii-small-teams-deliver-lower-cost-higher-quality

[5] http://www.qsm.com/process_improvement_01.html

[6] http://www.qsm.com/blog/2012/part-ii-small-teams-deliver-lower-cost-higher-quality

[7] Brooks, Frederick. The Mythical Man-Month. Addison-Wesley: Boston, MA. 1975.

[8] Scacchi, Walt. "Understanding Software Productivity." Advances in Software Engineering and Knowledge Engineering, Volume 4 (1995): 37-70.

[9] Sackman, H., W.J. Erikson, and E.E. Grant. "Exploratory experimental studies comparing online and offline programming performance." Communications of the ACM, Vol 11, Issue 1 (Jan 1968). pp 3-11.

[10] McConnell, Steve. Code Complete: A Practical Handbook of Software Construction. Second Edition. Redmond, WA: Microsoft Press. Section 28.3.

[11] DeMarco, Tom and Timothy Lister. Peopleware: Productive Projects and Teams. Second Edition. New York: Dorset House. p 47.

[12] Gould, John D. "Some Psychological Evidence on How People Debug Computer Programs." International Journal of Man-Machine Studies 7:151–82 (1975)

[13] McConnell, section 23.1.

[14] Scacchi.

[15] New York: McGraw-Hill, 1979.

[16] McConnell, section 3.1.

[17] McConnell, section 20.3.

[18] Myers, Glenford. "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections." Communications of the ACM 21, no. 9 (9): 760–68.

[19] Gordon, Scott V. , and James M. Bieman . "Rapid Prototyping and Software Quality: Lessons from Industry." Ninth Annual Pacific Northwest Software Quality Conference, 1991.