allan's blog - Agile & Digital Business

I help companies and teams that create software.

Thursday, August 04, 2016

Software development is upside down

In the software development world a number of common management maxims need to be reversed if one is to be an effective manager. Those who manage software development - and I include all the non-commissioned managers in this, Architects, Team Leads, Scrum Masters etc. - need to turn their thinking upside down.

Here are a few I spot all the time but I'm sure there are more:

1. Diseconomies of scale: larger teams are less productive per worker than smaller teams, larger code bases are more expensive to maintain (enhance, debug, change) than smaller code bases per unit of code, large requirement requests (documents) are more expensive per request than small requests and so on.
2. **Higher quality is faster**, there is no such things as "quick and dirty": delivering quality work is faster than delivering shoddy work because shoddy work needs fixing. Even if the first low quality piece of work does arrive more quickly, the second will be take longer down because the first gets in the way. Each shoddy piece of work will cost more because subsequent work will take longer.

Software product has many "qualities": functionality, speed of execution, usability, etc. etc. What constitutes quality varies from product to product however… all software products exhibit two qualities in common: number of defects (bugs) and ease of maintenance (maintainability). When I talk about quality it is these last two items I am talking about.

Whenever you find a high performing software team you find a high quality code base (low defects, high maintainability). Hardly surprising if you have read the work of Capers Jones and Kent Beck.

1. **Teams over individuals**: There are times when a lone developer who can sitting up all night and deliver a world beating product. Particularly at the start of a new technology: think Nick D'Aloisio writing Summly, Matthew Smith writing Manic Miner, or Dan Bricklin and Bob Frankston writing VisiCalc in two months.

We admire people like D'Aloisio, Smith and Bricklin but they are poor role models. Most serious software development is a team sport. The characteristics which make the lone developer successful are a disadvantage. Managers who believe in the Lone Hero Developer model do everyone a disservice.

The constraint in developing software is not typing speed, it is thinking speed. We need people who can share, who can discuss, who can work together. That is why Pair Programming can be far more effective than solo programming and why Mob Programming is on the rise.

1. **Doing is the quickest way of learning**: when processor cycles were expensive and difficult to get (think 1970, IBM mainframes, OS/360, IMS hierarchical databases and less than a dozen internet nodes) it made sense to think through every possible angle before developing something. Back then most systems were a lot smaller than they are today.

Today processor cycles are cheap, you have more CPU power in your pocket than Neil and Buzz took to the moon.

The quickest way to find out if a technology can do something, the quickest way to learn a technology, and the quickest way to find out what customers think is to just do something and see what happens.

There is a place for planning, but planning has rapidly diminishing returns. A little bit of planning is valuable, but a lot is counter productive: the return from lots of planning is minimal and it delays learning by doing.

1. Do it right, then do the right thing: modern development is inherently iterative. If a team can't iterate they cannot perform. If we are to learn by doing we must iterate: plan a little, do a little, review the results, plan a little, do a little….

   "Customers don't know what they want until they see it"

Or perhaps:

> "Companies don't know what will succeed in the market until they ask people to part with money."

Again and again we see that customers need to be shown something in order to understand what a product can do for them. And again and again we see that until a product is in the market and customers are asked to exchange money for it the true value, the ultimate "doneness" cannot be judged.

Only when we are in the market, only by showing what we have, can we refine our ideas of what is needed and what is valuable. And when we have this information it is through iteration that we act on it.

If the team can't iterate (do it right) then they have no way of learning and validating their plans. Sure, doing the wrong thing effectively is pointless, but the only way to find out what is right and what is wrong is to do something, anything, and iterate.

1. Worse is better: the famous Dick Gabriel from 1989:

   "the idea that quality does not necessarily increase with functionality—that there is a point where less functionality ("worse") is a preferable option ("better") in terms of practicality and usability. Software that is limited, but simple to use, may be more appealing to the user and market than the reverse."https://en.wikipedia.org/wiki/Worse_is_better

Sometimes creating a worse product is a more effective strategy than creating a better product. Building a better mouse trap is rarely the route to success.

To a business mind this maxim sometimes seems obvious but to an engineering mind it is shocking.

And this final maxim means that all of the above propositions are sometimes reversed.

(Many thanks to Seb Rose for comments on an earlier draft of this post.)

Posted by Allan Kelly at 10:32 am

2 comments

Add a comment as Allan Kelly

Top comments

Alfonso Burgos shared this via Google+ 11 months ago - Shared publicly

Ryan D via Google+ 11 months ago - Shared publicly
Are you developing software upside down?

· Reply

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

Zen'in keiei - more Japanese - every person a mana...

- ► July (4)
- ► June (7)
- ► May (4)
- ► April (4)
- ► March (2)
- ► February (3)
- ► January (4)

- ► 2015 (42)
- ► 2014 (41)
- ► 2013 (43)
- ► 2012 (45)
- ► 2011 (53)
- ► 2010 (70)
- ► 2009 (90)
- ► 2008 (85)
- ► 2007 (79)
- ► 2006 (77)
- ► 2005 (62)

JavaCodeGeeks

Syndicated to Java Code Geeks