Need to unlearn

Like many men I tend to like my old clothes. Unlike many I've come up with a strategy that helps me enjoy buying new ones: I clear out the wardrobe; rather ruthlessly I throw away things and make space. This done I know I need new clothes and can enjoy buying them.

I need to use a similar strategy when I write articles and patterns. Sometimes I draft a piece and there is a sentence, or even a paragraph which I really like, maybe its witty, sarcastic or makes a subtle side point, or maybe its an excellent example of something. So, I edit my draft - and I should mention I do a lot of editing - and as I edit I keep the chosen sentence. But over time it doesn't connect as well with what is around it and maybe I have to rewrite some of the surrounding text to lead up to this one sentence.

Eventually it becomes clear that this sentence is more of an obstruction than a support. It has to be chopped out so the rest of the text can make its point with brevity and clarity. It may be painful to do - like getting rid of those old clothes - but your better off without it.

(Actually, in truth, I usually use the same trick I do with old trousers. Take them out of the wardrobe and put it to one side somewhere, usually a box under the bed. If you don't need it in the next six months you aren't ever going to need it. So, I find files on my hard disc with little bits of articles which I never get around to using.)

The same thing is true in software. Sometimes a piece of code is so attractive I don't want to loose it - say its a nifty bit of template meta-programming, or a well formed class. No matter how nifty the code it can still restrain you, sometimes these things have to go for the greater good.

And it doesn't end with code. In fact, those who study these things would consider this *unlearning*. In the same way that we learn something we sometimes need to *unlearn* something. A solution that worked well in the past doesn't work well now. If we continue to rely on yesterday's solutions we stop ourselves from learning new things, like clothes our solutions come to look dated and full of holes.

The software development community could benefit from a bit more unlearning. While we're pretty good at dreaming up new languages and methods we're not so good at throwing some old ideas away. Sometimes our old ideas work to our benefit, they allow us to quickly diagnose problems and develop solutions because we've seen the problem before.

On other occasions these very short cuts work against us. We use mental models and assumptions that aren't valid for the problem in hand. Worst of all, we don't always know we're making these assumptions. When I was an undergraduate I had a lecturer who always told us to "Document your assumptions" problem was, I didn't realise that I was making assumptions. That's one of the problems we face, unconscious assumptions, how do we know we are making them?

Sometimes of course there are big red flags telling us to drop our assumptions. For example, when you change jobs, in a different company, with different people we need to change. Unfortunately its too easy to keep fighting the last war, or see our last employer through rose-tinted spectacles, your new colleagues don't necessarily want to hear about how good (or bad) the last place was.

Too often new people are encouraged to "hit the ground running" when they start a new job - especially if they are in a contract position. To do this denies employees the time to learn and to jettison some of the past and make a fresh start.

I've been guilty of this too, my blood starts the boil the moment I'm introduced to a "project manager", all these assumptions kick in: all they care about is GNATT charts, they believe estimates and the waterfall model, they want to divide, rule and micro-manage. I have to fight these assumptions, ask myself "What proof is there that this project manager is like this?"

Recognising and changing our assumptions isn't easy. It is especially hard when you try and do it on your own. Even looking data can be confusing, as we tend to see data that supports our point of view rather than refute it.

Writing in the Financial Times, Carne Ross (2005) described the how the British and American Governments argued at the UN with the French and Russia Governments about the 1991-2003 sanctions against Iraq. The two sides cited the same reports to support their case. Ross suggests that both sides where not guilty of ignoring the contradictory evidence, merely that they failed to see it. The assumptions each side made blinded them to contradictory data, they could read the words but their meaning was lost.

We often need other people to help us see our own assumptions; talking problems through helps us understand them and expose our assumptions. Other people come with their own, possibly different assumptions and we can all help highlight one another's assumptions. But, when we are locked in confrontation with others we become defensive, to admit an assumption, let alone change it would be to give ground.

The problem of incorrect and unspoken assumptions affects all aspects of software development: we think we know what the customer wants, or we think we know what the software design should be but sometimes we're wrong. The need to unlearn assumptions is particularly apparent when it comes to process and literature.

Although it's a great great book I'm getting a bit fed up of people citing Brooks' *Mythical Man Month*. It was written 30 years ago about a project that occurred 40 years ago. Haven't we moved on a bit?

While there is some great advice in Brooks' work there is some we need to unlearn. Lets start with "Build one to throw away, you will anyway." Brooks himself has changed his mind on this:

> " 'Plan to throw one away; you will anyhow.' This I now perceive to be wrong, not because it is too radical, but because it is too simplistic.
>
> The biggest mistake in the 'Build one to throw away' concept is that it implicitly assumes the classical sequential or waterfall model of software construction." (Brooks, 1995, p.265)

Then there are *Chief programmer teams*. This is the idea that we can have a few great programmers and arrange things to support them, keep them working productively and all will be right. This approach leads to teams where several lesser programmers work individually on minor pieces of functionality while the *Chief* or *super* programmer(s) delivers the real value. Consciously or unconsciously managers and programmers believe that Jim the super-programmer will deliver 70% of the project

while his three helpers will deliver 10% each, of course they'd like four super programmers but they "can't find them" so they settle for just reducing the load on Jim.

This is quite the opposite of what many people now think and flies in the face of what *Agile* methodologies advocate. Here - as in the much of twenty-first century modern business life - it is the team that is important. Whether it is serving fries in McDonalds, building a Nissan or writing software, simply, the scale of modern endeavourers means that it is the team that is the building block not the individual.

So, it is with dismay that I hear developers and managers proclaim, "If we only had a few more good people" or "Where can we get more good people?" It is not the lack of individuals that hold or developments back but the lack of good, productive, teams.

We need to apply a bit of unlearning here. Lets try and unlearn this particular mantra.

Sure, maybe one in 100 engineers is more productive than the other 100 put together but are we right to base our entire development process around finding this individual? We need to find them, hire them, motivate them and retain this one person. Even if we can do all that is it right to base an entire strategy around this person? And the chances are, one person isn't enough, we're going to need 10, 20, 100 more like him (and it usually is a him.) And how do these guys work as a team? Usually not too well.

It could be that our one individual is actually holding the team back. They may actually block others from dealing with a problem, or their very productivity may hide a fault with the team. Alistair Cockburn tells the following story:

> "a consultant who visited his company and said words to the general effect of, 'If I go into a company and see one super-salesman who is much better than all the rest, I tell the owners to fire this man immediately. Once he is gone, all the others improve and net sales go up'." (Cockburn, 2003, p.27)

Fact is, the super-programmer approach doesn't scale.

Instead, we need to hire and develop teams of people. We give them the tools they need to do the job, and we remove the blockages that stop them from working more productively. We encourage them to improve themselves, their environment, processes and the company - and that means we aren't scared to change, whether it be moving desk or trying a new way of working, we reject the assumption that tomorrow will be a repeat of today.

If we are to expose assumptions we need to enter into open dialogue with others - not necessarily people who hold the same point of view. We need to allow time for this, we need to understand our goals and share mutual goals. Above all we must be prepared to unlearn ourselves, if we start with a position to defend and assumptions we're unwilling to let go of then we aren't going to get very far.

Simple really. Well simple to say, unfortunately, it's incredibly hard to do, after our unlearning we need to learn again.

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

Cockburn, A., 2003 *People and Methodologies in Software Development*. PhD. thesis, Oslo

Ross, C., 2005, *War Stories*, Financial Times Weekend Magazine, 29 January 2005