SE and OL.doc

Software development and the learning organisation

"When you ask people about what it is like being part of a great team what is most striking is the meaningfulness of the experience.... Some spend the rest of their lives looking for ways to recapture that spirit." Peter Senge, 1990.

Think back over the last ten years, what have you learnt? What have we, the programmer community, learnt? On mass we've learnt C++, added the standard library, re-learnt ISO-C++, incorporated meta-programming and picked up Java, C99 and C#. And don't forget Python, JavaScript, Perl and other scripting languages that didn't exist in 1993.

Then add the technologies we've invented and learned: HTML, CGI, HTTP, ASP, JSP, XML, Or maybe you've avoided the Internet and just moved through Windows 3.1, 95, NT, 2000 and XP, or one of a myriad of Unix/Linux systems and versions.

The programmer community is constantly learning. If change is the only constant then learning is the only real skill you need. Indeed, how can we change if we can't learn?

Being a programmer you learn the technologies, but you also have to learn your "problem domain." That is, the field you are developing applications for. Some are lucky enough to work on IT technologies and develop e-mail systems, or databases, but most of us work outside the technology domain, so, I've managed to become a minor expert in train time-tabling, electricity markets, and financial instruments.

All the time we are learning, our teams are learning and our organisations are learning. So too are our customers who use our products. This happens whether we intend it to or not. Authors like Peter Senge and John Seely Brown argue that we can harness this learning through "Organisational Learning", so creating "Learning Organisations" which deliver better businesses - and in our case better software.

How does learning relate to software development?

If we look at the software development process there are at least four key learning activities:

- ? Learn new technology
- ? Learn the problem domain
- ? Apply our technology to the problem, the process of problem solving is learning itself
- ? Users learn to use our application and learn about their own problem which changes the problem domain

Each one of these points reinforces the others: in our effort to solve a problem we need to learn more about the problem, our solution may use a technology that is new to us. When the user sees the end product their mental model of the problem will change too. They too will learn, through the software, and acquire new insights into the task that may lead to changes to the software.

Learning is thus inherent at every stage of software development. We can either choose to ignore it and muddle through somehow, or to accept it and help improve the learning process.

Maybe your manager is having difficulty understanding this - after all he hired you because you already know Java so why do you need to learn some more? - so lets put it in business terms.

Although we can buy the fastest machines on the market, only hire people with an IQ above 160 and expand our teams endlessly and work to ISO-9001, we aren't actually doing anything our competitors can't do. All we are doing is proving we can spend money. Worse still, none of this guarantees we will actually develop good software.

If instead of viewing software development as a problem task we view it as a learning activity we get some new insights. First we can recognise that most developers actually want to make customers happy, what is more they like learning new things. It is just conceivable that a firm which encourages its staff to learn will find it easier to retain staff, hire new staff and at the same time see the ability of existing people increase.

Software development and the learning organisation

SE and OL.doc

Now to be really radical John Seely Brown and Paul Duguid believe that learning increases our ability to innovate. If we can learn better we will find we can produce new ideas and better solutions.

Since software development is intrinsically a learning process it doesn't seem that great a jump to claim recognising it as such, removing barriers to learning, and promoting learning within our group will improve our software. Once we do this we have something that competitor firms can't copy because we will create our own environment.

What can we do to promote learning?

The one thing I'm not suggesting is that you go to your boss and ask to be sent on a course. Brown and Duguid (1991) suggest there are two type of learning:

- ? Canonical learning: going on courses, sitting in class rooms, reading manuals
- ? Non-canonical learning: learning by watching, doing, listening and a whole bunch of other stuff.

What is more they go on to suggest that the second form, is the better. By learning non-canonically we are more flexible and innovative. While canonical, 5-day, ± 1500 courses have a use, there is a lot more things we can do to promote learning in our organisations.

Before we go further, it is worth pointing out that there are two types of knowledge. The sort we're all familiar with, explicit knowledge, which can be written down, codified. Go pick up your copy of Stroustrup, you can learn C++ from that, its all explicit knowledge.

The second form is subtler: tacit knowledge. This is more difficult to write down and we normally learn it by some process of osmosis. Much of this knowledge is embedded in our work environment or our programmer culture. So, when you write **del ete this** you aren't breaking any of the rules in Stroustrup, its legal, but all of a sudden you're in trouble because your team doesn't do that. Of course, coming from a COM shop you think **del ete this** is perfectly OK.

This is a simple example of tacit knowledge and the fact that I can write it down maybe invalidates it but I'm sure you get the idea. But how do we learn this stuff?

We get much of it through our society. That is, by watching other programmers, reading their code, exchanging war stories. Being programmers of course we like things to be black and white, quantifiable, written down, codified, so I'm sorry to tell you it ain't going to happen. In fact writing it down may make things worse!

In part we have so much knowledge we can't write it all down. Some of it is so obvious nobody thinks it worth writing down, and some we can't even put into words - although we may be able to mumble some grammatically incorrect phrases over a beer which sticks in someone's mind.

Tacit knowledge is the reason so many specifications are inadequate. The stuff is a lot like jelly, you can't nail it down, it isn't in the specification because it is hard to codify. Only when you come to use the specification do you find gaps that are hard to fill. Computer code is inherently explicit knowledge.

Acquiring and learning to use this knowledge can take time. We need to be immersed in the society and let this stuff lap around us. Eventually we may try and do something, and from this we learn some more.

This brings us to another important point about this kind of learning. We're going to make mistakes. There will be failures. We need room to try ideas, see how they work, or don't work and add that information to our mental models. If we don't make mistakes part of our internal model will be missing.

For example, have you ever needed to write a piece of code and thought "I bet I could use templates for that? But I don't really know enough about meta-programming, O, I'll give it a try." And although it takes longer at the end of the week you have something that works and - very importantly - you understand templates? Along the way you probably made a million syntax errors, many compilation errors and got umpteen unexpected results but in doing so they completed your mental model.

Now the difficult bit for managers is to accept this trail-and-error approach. We need it. And although it doesn't look good when you're filling in the weekly timesheet the numbers are hiding the learning that occurred during development.

SE and OL.doc

So, we need to accept mistakes will happen, we need to accept that risk. And maybe we need to do away with processes that penalise taking risks and making mistakes. (Although you may not want to take risks the night before release.)

By implication here there needs to be trust. If we don't trust someone to do the job they won't feel able to make those mistakes. They'll stick to the tried and tested solutions. We need to trust them to know when it is OK to play and when they should focus.

Nor does learning finish with our own team. The QA team will be learning how to use our releases, and the best way to test them. And when we finish and throw the software over the wall, users will start their learning.

What should we not do?

Even those of who have worked in adversarial, deadline driven environments have learned. The first thing we need to stop doing is denying that learning happens. Brown and Duguid point out that when managers deny that learning is occurring two things happen:

- ? Individuals feel undervalued, managers don't recognise the role they play
- ? Managers think their systems are working, "We sent them on a Java course and now they all write quality Java" when in fact everyone is helping everyone else.

These effects describe "Plug compatible programmer" syndrome. Management feel, or suggest by their actions, that they can "just hire another C++ contractor" to plug a programmer shaped gap. After all, all C++ programmers are compatible. Meanwhile, developers know you need more than just C++ knowledge to work on the system, so they feel their skills aren't recognised by management and leave.

Brown and Duguid also suggest that seeking closure can be damaging too. By seeking closure, say by writing up all that is known about a given application, complete with UML diagrams, we actually inhibit future change.

I once worked on a team who worked to ISO-9001 standards. You weren't supposed to change the code without changing the documentation. Not only did this increase the work load but it made you reluctant to change anything. Increasingly code and documentation said different things and you had to talk to people, or step through the code with the debugger to see what was happening, that is, exactly the situation the standard was meant to prevent!

The need for closure made things worse. This happens all the time with documentation, whether it is program documentation or specifications. The emphasis on closure is one of the fundamental reasons waterfall methodologies are so troublesome.

Closure prevents change and it prevents further learning, but change and learning will happen. This doesn't only happen with us, the developers, it happens with our customers. A customer signs off on a spec, we develop the UI, show it to the customer who now wants to change it. In seeing the UI the customer has learnt something. Customer and developer are both engaged in a joint learning process.

This search for closure manifests itself in many forms: product contract, specification, program documentation, working procedures, code standards and perhaps the worst of all: code itself!

Some degree of closure is always necessary, otherwise we would never make a release. However premature closure limits learning opportunities and development. We need to strike a balance.

Likewise, we need to strike a balance on how much risk we accept. One organisation I know introduced procedures asking for every change to be estimated in terms of lines of code. Developers would naturally over estimate but it also made them more risk averse, there was clear message: management wanted change and risk limited. Thus they limited learning opportunities - specifically code refactoring.

Blunt measurements such as lines of code, and timesheets asking what you do with every half-hour in the week also send another message: you aren't trusted. These are tools of managers who believe that software development is an industrial process. "Scientific management" is at odds with the concept of learning because it doesn't allow for learning and change. It assumes that one person knows best, the manager, the designer, or the architect has looked at the problem and found the one true solution.

SE and OL.doc

Practical things to do

Organisational learning isn't a silver bullet, you can't go out and buy it from Rational or Accenture. It is a concept. One of the ways it manifests itself as a *Learning Organisation*. You have to build this for yourself. The good news is that it need not entail a lot of up front expenditure, no expensive courses from QA or Learning Tree.

Much of the change is about mindset: accept mistakes will happen, accept some knowledge is tacit, accepting change, trusting people and leaning to live with open issues.

Managers face a difficult position. They can't force anyone to learn, nor should they. However, there are some practical things they can do to encourage the process. These have both an obvious role to play and a less obvious: by arranging these events and making time for them there is a message that "it is good to learn."

What ever your position you need to start with yourself. For an organisation to learn the teams making up the firm must learn, for teams to learn individuals must learn. And we must learn to learn.

If you want to start encouraging others here are a few things you could try:

- ? Set up an intranet and encourage everyone to produce their own web pages, even if these aren't directly work related.
- ? Keep documentation on the intranet in an easy to access format, e.g. if your developers are using Solaris don't put the documents in Word format.
- ? If your documentation must live in a source control system then arrange for it to be published to the intranet with the batch build.
- ? Allow an hour or two a month for "tech talks" get developers to present work they have done on the project or outside the project.
- ? Encourage debate friction can be a good thing.
- ? Organise a book study group.
- ? Make your procedures fit your working environment and be prepared to change them.
- ? Hold end of project reviews, Alistair Cockburn (2002) even suggest such reviews should be held during the project why wait to the next project to improve things?

Finally

I've only scratched surface of this subject, I'm still learning a lot about it myself but I think it has important ramifications for the software development community.

Unfortunately these ideas really require support from management before they can really deliver benefits, and I know most Overload readers are practising programmers who regard managers as part of the problem not the solution. That's why I spent some time advocating organisational learning in language they may understand.

Still, there is a lot we as a community can learn here and most of it has direct applicability to software development on a day-to-day basis.

Bibliography and further reading

John Seely Brown and Paul Duguid, 1991: "Organizational learning and communities-of-practice: Toward a unified view of working, learning, and innovation", Organisational Science, Vol. 2, No. 1, February 1991, also http://www2.parc.com/ops/members/brown/papers/orglearning.html

I've drawn extensively on this article, although it is quite long (18 pages) it is well worth a read.

Brown, Collins & Duguid: Situated Cognition and the Culture of Learning - http://www.ilt.columbia.edu/ilt/papers/JohnBrown.html

Software development and the learning organisation

SE and OL.doc

More psychological than the first but still interesting. Brown is a PARC researcher and has several interesting papers at http://www2.parc.com/ops/members/brown/index.html - including some are on software design.

Cockburn, A., 2002: Agile Software Development, Addison-Wesley, 2002

I haven't heard anyone from the Agile methodologies movement specifically link them with Learning Organisations but I instinctively feel they are. (Hopefully I'll explore this some more in future.) In the meantime you'll find terms such as "courage" and "coaching" used in both sets of literature, and similar discussions on the importance of people, teams and listening.

Nonaka, Ikujiro, et al., 1995: The Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation, Oxford University Press, 1995

Nonaka distinguishes "Knowledge Creation" from organisational learning but the two are complementary. As the title suggests, this book concentrates on the way Japanese companies exploit knowledge creation.

For a book summary see: http://www.stuart.iit.edu/courses/mgt581/filespdf/nonaka.pdf

Senge, P.M. 1990: The Fifth Discipline, Random House, 1990

Easy to read, authority introduction to "The Art and Practice of the Learning Organisation." To a hardened software engineer this may look like a touchy-feely meander. Don't let this put you off, you can't get the most from people if you stick with a binary view.