Ruminations on Knowledge in Software Development

In computing we are accustomed to shunting bits and bytes about. We call this data, we may even accept this represents information, but is it knowledge? In fact, are there any real and important differences between data, information and knowledge? And are these differences of any importance to us when we develop software? (And, with all these questions, am I in danger of turning into a character from a well know HBO series set in New York?)

This article continues the theme of learning from my previous Overload piece, "Software Engineering and Organisational Learning." In part you may like to consider this an simultaneous review of several books which promote the same ideas.

The difference

In everyday language data, information and knowledge tend to be interchangeable terms. Certainly, most dictionaries I've looked at seem to define each term in terms of others. However, if there is no difference between these terms what is the point of having them?

For their book, *Working Knowledge*, Davenport and Prusak (1998) noted that there are many words and definitions that are applied to the nebulous ideas of data, information and knowledge. But since we have enough trouble defining just three terms we had best not ponder on too many. Using their working definitions we get:

? Data claims to be some objective facts about events.

? Information is a message intended to change the receivers perception of something, it is the receiver rather than the sender who decides what the message means.

? Knowledge is a fluid concept, incorporating experience, values, context that exists inside an individuals mind or in the processes and norms of an organisation.

One of the leading writers on the subject of knowledge is Ikujiro Nonaka, he attributes (1995) three attributes to knowledge:

? Knowledge is about beliefs, commitment, and is a function of perspective and intention

? Knowledge is about action

? Knowledge, and information, are about meaning and is context specific.

Later, he extended these ideas to place knowledge within a concept called "ba" (1998). This is a Japanese term us uses to describe the space in which knowledge exists, take away "ba" from knowledge and what you are left with is mere information.

For example, Meyers *Effective C++* is nothing more than a list of 50 items in strange bizarre language - at least when Nick Hornby publishes a list there are a few laughs. But add experience of C++, the values of the C++ community and the fact that readers are usually practising C++ programmers and suddenly the contents of *Effective C++* take on a different meaning.

Another example of "Ba" occurred during the development of Concorde. The Soviet Union decided it had to have a supersonic passenger plane to rival the Anglo-French Concorde and the proposed Boeing 2707. Lacking the time and expertise the Soviets stole the blue prints of the plane and set about building their own Koncordski, the Tupulov 144. When revealed the plane looked like Concorde, and it even flew but it didn't perform as expected.

Although they had the plans the Soviet engineers lacked context and culture of the designs. Measurement systems where different, ways of working where different, and notations where different. Thus, they weren't about to build an exact replica of the Anglo-French plane.

Similar things happen to software project when a new team takes over an old project. The project code may come with documentation and UML charts but it is still difficult to understand. The new team lack the "ba" of the old team. This may explain why developers tasked with maintenance often feel the need to re-write existing code.

Where is knowledge in software development?

The whole software development process is an attempt to codify knowledge. We start with some vague idea of what a system should do and, through successive processes of specification, design, implementation and testing, try to turn that knowledge into a working, useful, model.

Our problem is that knowledge is difficult to codify. As software developers our skills and knowledge reside in our own domain, our own field of "ba". We take a problem domain, with its own "ba" field and attempt to produce a product which will exist in both domains, satisfying the requirements of the problem domain while meeting the engineering requirements of our own solution domain.

Software needs to exist simultaneously in these two environments. Commercially it is the part seen by customers that tends to get priority, even though this represents the tip of the iceberg (Figure 1). As engineers we see the bigger, more complex problem underneath the waves.

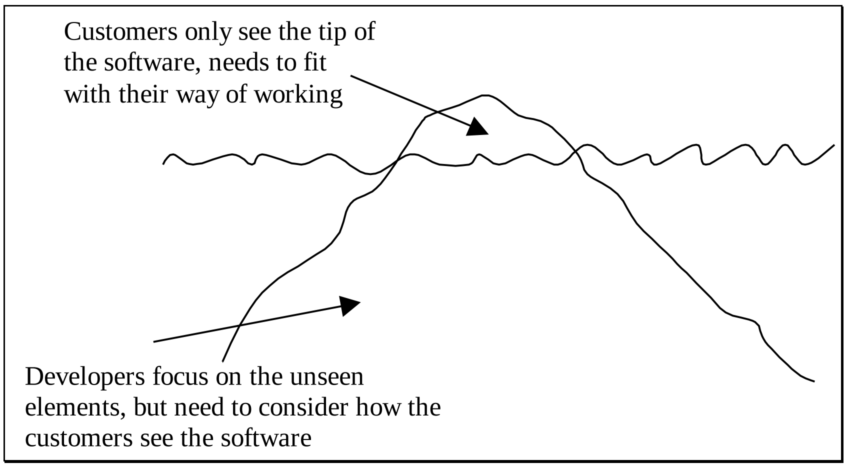
Customers only see the tip of the software, needs to fit with their way of working
Developers focus on the unseen elements, but need to consider how the customers see the software

Figure 1 - Software is like an iceberg

Codification

As if this weren't enough, much knowledge is actually tacit. That is, it is uncodified, it is not written down anywhere. We may not realise we have this knowledge until we attempt to write it down or do things differently. Usually it is just "the way we do it around here."

When we deliver a program it enters into the users domain. It becomes has to live as part of their "ba" so we must respect what users know and expect. If we embed values and judgements into our software which are different to the ones in common use our customers will find the system counter-intuitive and difficult to use. If, on the other hand, we tailor our system to their norms they will find the system easier use.

Of course, often the whole point of introducing software is to disrupt current practices so they can be changed. However, we should be sure we know which practices we are attempting to change and which we want to keep. There is no point in introducing software which forces doctors to measure temperatures in Kelvin if we are trying to change their prescribing practices.

Specification

It is when we come to write the specification that we start to grasp the difficulties that are presented by both "ba" and tacit knowledge. Specifications have a tendency to grow like Topsy, they never seem to be complete. If we attempt to write a complete specification we must not only codify the system requirements but also the context, the "ba" they exist in. To be fully complete the specification for the prescribing system would need to explain what temperature is, how it is measured and what the units are.

Specifications are themselves abstractions, and in making the abstractions we have to leave out detail. But the attempt to leave out detail leads to incompleteness because we rely on context to provide it. It is always possible to add more explanation to a specification. Thus we end up with thousand page specifications.

Secondly, our specifications still haven't tackled tacit knowledge. As we write the specification we will uncover more and more undocumented rules of thumb, methods of working, common practices and so on. This continues as the system moves to implementation and we see how the different bits interact. Testing, almost invariably, throws up undocumented assumptions, missed function points and incompatible implementation.

Hand-over

Anyone who has ever worked on a serious software system will have been involved in project hand-overs where on developers attempts to dump the contents of their brain, their knowledge, to a new team member. This can be scary if your arriving on the team and suddenly trying to absorb a million and one facts about a system, and if your the one trying to pass on the information - particularly if your leaving the company.

Documentation is of limited help. Like many developers I've experienced the mountain of documentation which lies in wait when you join a new project. Because it has been written down managers expect that simply reading it will make you as knowledgeable as the writer.

Again we see tacit knowledge and "ba" at work. The documentation can't possibly contain every thing the last developer knew about the system. Even if they divided their time equally between documentation and coding there are assumptions that will never make it to paper.

And reading the documentation when you first join a project means your reading it in the abstract. Until you have been emerged in the project, spoken to other developers - tried to understand the problem and the solution - large parts of documentation are meaningless.

Knowledge creation

In producing a solution to a problem we need to create new knowledge about the process and about the solution. If we understand the knowledge creation process it should help us work with the process rather than against it.

In writing about knowledge, Nonaka, proposes a four stage model (Figure 2) that turns tacit knowledge into explicit knowledge, combines it with other explicit knowledge and turns it back into tacit.

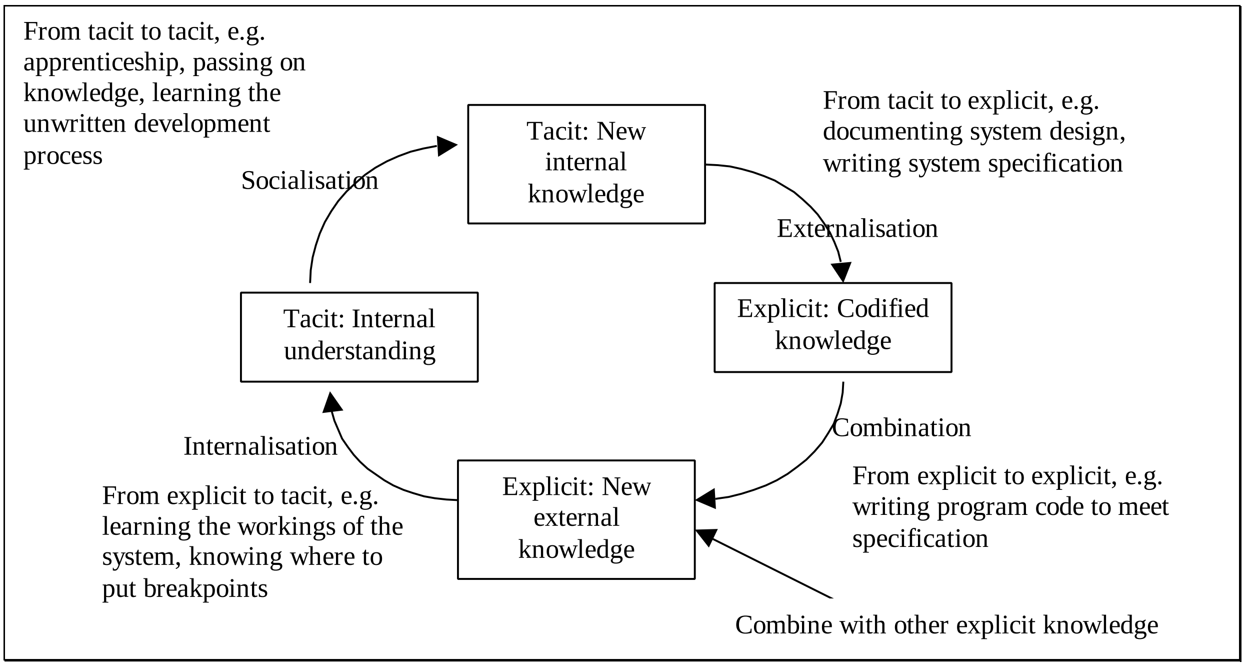
From tacit to tacit, e.g. apprenticeship, passing on knowledge, learning the unwritten development process
Socialisation
Tacit: New internal knowledge
From tacit to explicit, e.g. documenting system design, writing system specification
Externalisation
Explicit: Codified knowledge
Tacit: Internal understanding
Combination
From explicit to explicit, e.g. writing program code to meet specification
Internalisation
From explicit to tacit, e.g. learning the workings of the system, knowing where to put breakpoints
Explicit: New external knowledge
Combine with other explicit knowledge

Figure 2 - Nonaka's four modes of knowledge conversion (adapted from Nonaka, 1995, p.62)

With each conversion knowledge is extended. This may mean it is combined with some other knowledge to create new knowledge, or it may mean that more people understand the knowledge, it may also mean that individuals have a better understanding of the knowledge.

Just do it

Another of Nonaka's point was that knowledge implies action. We need to act on information in order for it to truly be considered as knowledge. After all, how many times have you written a piece of code which you know violates some best practice, but, for what ever reasons, time, laziness, expediency, you write it some other way? You have the information to write it better but you choose not to.

Software developers are not alone in this. Newspapers regularly publish stories about reports written for companies or Governments that are not acted on, how a study recommended X in 1998, and in 2001 Y happened because X hadn't been done.

In fact, there is a whole book on subject called - the *Knowing Doing Gap* by Pfeffer and Sutton (2000). They suggest that individuals, teams, and companies often know what the best thing to do it, but they fail to act on what they know for a variety of reasons. As well as discussing these problems Pfeffer and Sutton examine a number of companies who have succeeded in overcoming these problems and have enjoyed considerable business success.

One of the companies described in *Knowing Doing Gap* is SAS Institute of North Carolina. SAS is the worlds biggest privately owned software company - proof, if it was needed, that these concepts are applicable to software development.

Perhaps surprisingly Pfeffer and Sutton suggest that successful companies don't have any special secret ingredient, or magic bullet, they don't necessarily do anything other companies don't know about. What these companies *do do*, is to actually act on what they know. Simple really.

What do we do now?

Many problems in software development are of our own making. We don't do what we know to be right. We use myths to stop us acting on our knowledge, we get involved in infighting and, in many cases, we collude to support a system that we know could be better.

For example, the myth that the 1,000 page specification describes everything that we need to know. No serious software developer really believes this myth but people still contract to develop software on the basis that the specification contains everything we need to know. There is no silver bullet here, the solution is to stop propagating the myth and instead institute working practices that allow for learning and knowledge creation as we go.

Another myth particularly popular among managers is that of the *plug compatible programmer*. The idea that if a C++ programmer quits we can just hire another C++ trained developer to take their place. I can hear agreement from Overload readers as I write this. However, we developers must bear some of the responsibility here. IT people are known for changing jobs frequently, by doing so we propagate the myth that we can "hit the ground running" and plug an hole quickly.

This myth includes contractors and consultants - the hired guns of the industry. Managers believe they can hire consultants for a short-term role and let them go at a moments notice. Consultants like this myth because it leads to bigger pay packets and "freedom". But after a while we find managers dependent on contractors and only willing to hire those who have worked in similar roles already. Meanwhile, contractors complain that managers treat them like commodities and don't give them a chance to do something different.

I've been as guilty of this as anybody else. It can be financially rewarding way to work, and it seems to suit many individuals, and companies like the idea too. However, it leads to an inherent short termism and propagates the *plug compatible programmer myth.*

In both cases the process and the product are inherently linked. We shouldn't be surprised by this, processes are created to achieve goals. The problem is that just saying a process is there to achieve "quality" or "on time delivery" does not mean it will. Our processes are far more complex and can produce results we don't desire.

This isn't anything new, this is just another way of stating Conway's law (1968): organisations will produce software which is a copy of its own internal processes. If we want to produce good software, and help our employers succeed, we need to look beyond the immediate issues and see how all the pieces fit together.

Conclusion

Considering software development as learning and knowledge creation highlights the fact that it is difficult to communicate and codify what we want from a piece of software - the old "do what I want, not what I say" syndrome.

While software is key to "information economy" and used by "knowledge workers" we should consider software development itself as knowledge creation. The software development community tends to look inside for answer to problems, but there is much we can learn from elsewhere. The writers quoted here aren't specifically interested in software developers but their ideas are highly applicable. Just don't expect technical solutions, these aren't technical problems so there is no technical fix available.

Everything software developers do concerns the application of knowledge and learning. From specification, through design to delivery we are concerned with using knowledge and developing products from the application of our existing knowledge and the creation of new knowledge. Understanding this should help improve the development process.

Bibliography and further reading

Conway, M. 1968: *How do committees invent?*, Datamation, April 1968

Davenport, T.H., Prusak, L., 2000: *Working Knowledge*, Harvard Business School Press, 2000.

Kolb, D., 1976; *Management and the learning process*, California Management Review, Spring 1976, Volume 18, Issue 3.

Nonaka, I., Hirotaka, T., 1995: *The Knowledge Creating Company*, Oxford University Press, 1995

Nonaka, I., Konno, N. 1998: *The Concept of "Ba"*, California Management Review, Spring 1998, Vol. 40, No. 3

Pfeffer, J., Sutton, R., 2000: *The Knowing-Doing Gap*, Harvard Business School Press, 2000.

WBGH, 1998: *Supersonic Spies*, Nova, transcript at http://www.pbs.org/wgbh/nova/transcripts/2503supersonic.html

> The UK Channel 4 program *Equinox* is the US PBS program *Nova*, in 1998 a programme covered the development of the Tu-144, the Soviet Unions version of Concorde. A transcript of the programme is available from the US PBS site. The UK programme may have been slightly different but the substance was the same.