Lean, Agile

Paper for Workshop "The Software Value Chain"
OT2004
David Harvey, January 2004
Revision: 43

http://www.davethehat.com

A Brief History of Lean Thinking

In the years of reconstruction following the Second World War, Japanese industry (in general) and Toyota (in particular) had a problem, which was this: how to rebuild a shattered manufacturing base without recourse to either the huge market or the economies of scale available to Western (specifically US) companies, and in the face of severe credit restrictions imposed by the occupying forces (leading to reduced sales and very limited resources to invest in the new plant that everyone thought essential to efficient manufacturing).

At first glance, the task might have appeared hopeless. The image of the relentlessly efficient production line, with huge and specialised machines producing countless numbers of components for rapid assembly by teams of workers, was ingrained in business thinking. Toyota was nearly bankrupt when Taiichi Ohno, the company's Assembly Shop Manager, took in hand the task of redesigning production.

With the constraints in which he found his organisation, and drawing on practices which had been tried with success in other industries in Japan, Ohno's redefinition of production was – on paper – clearly focussed on getting the best out of limited investment:

- Build only what is needed
- Eliminate anything which does not add value
- Stop if something goes wrong

Build only what is needed – that is, build only that for which you already have a customer. This means you don't have to guess at demand, or hold expensive stock that ties up money and space while it hangs around. This applies at all stages in the manufacturing process: any step in production should build what is needed by the next step, not produce large batches of inventory that wait to be used.

Eliminate anything which does not add value – which means taking a hard long look at what you and your customer mean by "value", and making sure both that you're not wasting effort on activities which make unnecessary things, and that you're not doing anything to inhibit the flow of this value.

Stop if something goes wrong – if you're only building what is needed at each stage, you're able to identify a defect very close to the point at which it arises (in production systems organised around large batches, defects can go unnoticed for days or weeks). Take advantage of this by addressing the cause of the defect immediately. This is the

fundamental principle of zero-defect manufacturing, based on rapid feedback rather than inspecting the hell out of everything, which has been consistently misunderstood by software methodologists.

The Toyota Production System is also grounded in a set of values enshrined in a philosophy of work that

- respects those engaged in the work
- strives for full utilisation of workers' capabilities
- places authority and responsibility for the work with those doing it

The most startling manifestation of these values was seen in the authority of any worker on the line to halt the entire production process if they found a defect.

These ideas revitalised Toyota (and led to Ohno's rapid rise in the company – he eventually became Executive Vice President in 1975). While the ideas were common currency in Japan from the 1950s, they did not make an impact on Western industry until the 1980s. Ohno, now retired, had started writing about the Toyota Production System he had pioneered. More significantly Toyota and other large Japanese companies expanded in the 1980s to set up new manufacturing centres in Europe and the Americas. In motor manufacturing, these companies' ability to rapidly develop car models for a Western market that was having to re-adjust following the oil crises of the 1970s saw them overturn the dominance of the local industrial giants Ford and GM. The ideas found ready acceptance in fields outside manufacturing: logistics and distribution (for retail and mail-order) have also been revolutionised.

Books that Changed the World

The experience of Lean Production in the US was documented by James Womack, Daniel Jones and Daniel Roos in their 1990 book *The Machine that Changed the World*[1] and by Womack and Jones in their 1996 sequel *Lean Thinking*.[2] The earlier book arose from the International Motor Vehicle Program (IMVP) at MIT, a 5-year project investigating the role of the automobile industry in the world economy. The second book generalises the lessons learned from the IMVP programme, and describes experiences of introducing lean principles in a variety of industrial and commercial settings.

In *Lean Thinking*, Womack and Jones identify five principles behind the lean organisation of production or delivery:

- Specify value
- Identify the value stream – line up activities which contribute value, eliminate those which add no value
- Create the conditions for value to flow smoothly through the stream
- Have the customer pull value from the stream

[1] James P Womack, Daniel T Jones, Daniel Roos, *The Machine that Changed the World*, New York, 1990

[2] James P Womack, Daniel T Jones, *Lean Thinking: Banish Waste and Create Wealth in Your Organisation*, New York, 1996

- Pursue perfection – work on improving the responsiveness of the production system to the customer demand for value

While even in the mid-1980s other western writers had already started to describe the organisational theory and practice imported by the Japanese, these books were pitched at, and taken up by, a more general corporate audience across many industries. Daniel Jones formed the Lean Enterprise Research Centre[3] in 1994 in his native Cardiff, Wales, and through participation in several high-profile UK and European industry and government initiatives has been instrumental in spreading lean thinking and practices on this side of the Atlantic. In the US, Womack founded the Lean Enterprise Institute[4] in 1997, a non-profit research, publishing and training organisation dedicated to furthering the growth of a lean community across diverse business areas.

Software – the Road to Agile Development

Rewind now to the 1970s and early 1980s, where the phenomenal growth in capability of computer hardware and (at this time) the steady increase in the connectivity of systems was matched by a corresponding growth in the ambition of software designers, and hence complexity of the systems being developed. This growth in ambition outstripped the growth of tools to support the development of such systems, so in order to produce larger and more complex systems, larger teams were needed. It was clear that some principles needed to be applied to the organisation of these large teams and projects, and only natural that people concerned with development process looked to what was then regarded locally (at least in the US and Europe) as best practice. Hence the rise of systematic development processes which mirror the product development and manufacturing processes of contemporary industry, specifically with emphasis placed on requirements engineering as a model of specification, and inspection with respect to those specifications as a model of quality assurance. Taken to extremes, this leads to the extended product lifecycles experienced by the US car manufacturers prior to the lean revolution: requirements transferred as a large single batch to development, the system delivered as a large single batch to QA and production. Somewhere along the line, this process of inspection (of requirements, code, system) and formal sign-off was claimed to be the effective model of 'zero-defect' development: actually this way of proceeding is pretty much guaranteed to produce defects in the face of any level of complexity or uncertainty in specification.

With the rise of the object paradigm from the mid-1980s, it seemed at last that technology was providing us with some tools for managing the complexity of the systems we wanted to build. However, at the same time, the requirement for systems connectivity experienced an explosive growth (in retrospect, a singularity of sorts[5]) as a result of the phenomenon of the World Wide Web (itself predicated, amongst other things, on the ubiquity of increasingly powerful and low-cost personal computers). So while we now have programming languages and software tools beyond the dreams of the earliest developers, we have still, effectively, been playing a catch-up game.

[3] http://www.cf.ac.uk/carbs/lom/lerc/
[4] http://www.leaninstitute.org
[5] The concept in this sense introduced by mathematician and author Vernor Vinge in 1993. See http://www.ugcs.caltech.edu/~phoenix/vinge/vinge-sing.html

Enter Agile Development. When the history of Agile is written in years to come, two things will, I think, stand out. Firstly, the way in which any number of practices and principles long recognised as effective ways of working to deliver software were brought together largely independently by a number of separate groups and individuals in a striking example of convergent evolution. To identify a handful:

- Iterative delivery (described comprehensively by Tom Gilb in 1988[6])
- Patterns (motivating a desire to base how we work on what's seen to be effective *in the specific circumstances*, not what is theoretically correct or works in a parallel but not necessarily close environment)
- Work on organisational patterns, specifically by James Coplien, who first described pair programming in the wild[7].
- Focus on team and individual interactions (Gerald Weinberg's *The Psychology of Computer Publishing*[8] was published way back in 1971: more recently Tom deMarco and Timothy Lister's *Peopleware*[9] has had greater impact.

Secondly, even with hindsight a historian is bound to ask the question – what took us in software so long to get our act together?

Agile and Lean

Kent Beck famously addressed the frustrations of many with software development in 1998s in the following meditation:

> Software is too damned hard to spend time on things that don't matter. So, starting over from scratch, what are we absolutely certain matters?
>
> 1. Coding. At the end of the day, if the program doesn't run and make money for the client, you haven't done anything.
> 2. Testing. You have to know when you're done. The tests tell you this. If you're smart, you'll write them first so you'll know the instant you're done. Otherwise, you're stuck thinking you maybe might be done, but knowing you're probably not, but you're not sure how close you are.
> 3. Listening. You have to learn what the problem is in the first place, then you have to learn what numbers to put in the tests. You probably won't know this yourself, so you have to get good at listening to clients - users, managers, and business people.
> 4. Designing. You have to take what your program tells you about how it wants to be structured and feed it back into the program. Otherwise, you'll sink under the weight of your own guesses.
>
> Listening, Testing, Coding, Designing. That's all there is to software. Anyone who tells you different is selling something.[10]

[6] Tom Gilb, *Principles of Software Engineering Management*, first published Boston MA, 1988
[7] http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns?DevelopingInPairs
[8] Gerald Weinberg, *The Psychology of Computer Programming*, first edition New York, 1971
[9] Tom deMarco and Timothy Lister, *Peopleware*, first edition New York, 1987
[10] http://c2.com/cgi/wiki?ExtremeProgramming

These are the roots from which Extreme Programming grew, the core around which the XP values and practices coalesced. The thinking is strikingly parallel to Ohno in his redefinition of industrial production.

Lean Software Development is now the title and subject of a book by Mary and Tom Poppendieck[11] (significantly, Mary combines experience of software and systems development with industrial production management and product development). The book is organised as an "agile toolkit", and constitutes both an examination and justification of agile practices in terms of lean thinking, and a recipe book of techniques for organising and executing software projects. The book explores the many correspondences between lean and agile in far greater detail than is possible here: for now I will consider a subset of XP practices in the light of lean thinking (I've chosen XP practices because at present they are the most crisply defined and articulated: by definition, any set of practices established by a team working in an agile fashion should support the same conclusions).

For many years *inspections* were promoted as a powerful technique for achieving quality in software deliverables. Specification, design, code, documentation, all could be reviewed through a formal inspection process which would mean that the end product would (surely) meet quality and functional requirements. This model of quality assurance has failed in most software projects because

- It is not responsive enough. Inspections by definition are after the event, organising inspection meetings, reviewing, reworking and reviewing the rework all add time to the critical path of a project
- There is too much to inspect. Inspection methodologies try to counter this by specifying the size or amount of material to be inspected, however to inspect everything in this way leads to many more inspections
- People are too busy to properly review the material
- In general in software the standards by which we might inspect and qualify the artefacts of a development process are poorly defined

Clearly there are some life-critical systems in which this level of oversight is highly desirable, whatever other processes are involved in delivering the software. Clearly for most of the systems we develop, some other way is needed.

The systemic nature of the XP practices make it hard to single out one or a handful with a particular goal in mind – all the practices can be said to address fundamental issues of quality (delivering the right thing to people who want it, in a timely fashion, and in a form that's usable). The practices together render unnecessary inspections of the type that used to be common in software projects:

- Because of the planning game and the on-site customer, the project always knows what features need to be developed
- Because of frequent delivery and the planning game, it doesn't matter if up front all of the features required aren't known or even envisaged. Detailed specification happens close to the point of development

[11] Mary and Tom Poppendieck, *Lean Software Development*, Boston, MA, 2003

- Because of frequent delivery the customer gets to see real functionality shortly after the point of detailed specification of that functionality – no-one has to review a specification document or a software design
- Because of pair programming, software delivered to integration contains many fewer defects (firstly, because the quality of concentration and effort in effective pairing leaves less room for accidents, secondly because defects are spotted and addressed at the point of coding[12])
- Because of test-first programming and continuous integration, any defects that do get through are identified very close to the point at which they are introduced.
- Because XP promotes coding only what is needed to meet the stories for the current iteration, functionality does not lie around unused and untested by its users for long periods of time.
- Because iterations are frequent, if there is an issue with the feature delivered it is identified by the user close to the point at which it was developed
- Because XP deprecates the production, review and approval of second-order artefacts, everyone has more time to devote to the real job at hand, which is delivering features in software.

The contrast between an inspection model for quality and the XP practices is exactly that between batch-mode inspection-monitored production and the lean production model:

- *Build only what is needed* – customer stories for an iteration constitute the *pull* of value from the development process. Delivering on the stories means that the project is not batching features for testing and use in a more distant future.
- *Eliminate anything which does not add value* – enshrined in XP mantras like "the code is the design"
- *Stop if something goes wrong* – defects are spotted and addressed at the point of introduction in pair programming, and very close to this point due to continuous integration. As in Toyota, anyone has the authority to stop the production line.

Why it Matters

We've all become so used to talking about the software crisis that the concept no longer seems to have any strong effect. Perhaps the point is that there is no crisis – after all, following the fears around the millennium we're still all here, the threats to civilisation as we know it stemming from issues less tractable than the number of bytes used to store a date. Organisations continue to start, thrive, disappear, software gets written, shipped, used.

Yet there are two crises which software development faces. One is economic, the other, less tangible but no less real, is an identity crisis.

Firstly, the economic argument. Increasing globalisation combined with the connectivity of the Internet has blown open the market for software developers, the

[12] This is increasingly supported by research into pair programming. See for example Laurie Williams et al, 'Strengthening the Case for Pair Programming', *IEEE Software*, July/August 2000 (online at http://collaboration.csc.ncsu.edu/laurie/Papers/ieeeSoftware.PDF)

trend for development organisations (software houses or large end users alike) under cost pressure to outsource or relocate development to low cost locations shows no sign of slowing. Cost pressures lead large organisations and individuals who use software to search for cheaper sources of the software tools and infrastructure they require, increasing the downward pressure on software costs. The consumer technology market itself is changing, with integration of specialised hardware devices (PDAs, phones, cameras, music players) distracting us from the fact that there is less and less innovation in software products.

The options for an organisation involved in developing software boil down to two

- Continue doing what we're doing, but for lower costs. I don't like this option, for two reasons. Firstly, it means we're either paid less for the same work, or our employment disappears altogether as it's contracted out or outsourced. Secondly, it enshrines the current level of expectation of delivery and performance in software delivery, when I think we could all be doing so much better
- Radically improve quality and productivity by adopting better ways of working to deliver software. This is the theme of this article, and if you're still with me I hope you'll agree that this is the way we need to go

Software's second crisis is its identity crisis. We've been told that computing is a science, that it's engineering, that it's manufacturing, that it's architecture, and that it's different from all of these[13]. There is indeed a point at which all these metaphors break down (and some have proved more useful than others) – however, while software may be different, the *people* who develop software are still – well, people, as much as it may flatter our egos to think our skill in software sets us apart from the common crowd. We may simply have been looking at the wrong thing for all of these years, mixing discipline-envy of those professions which seemed to us to be so much more self-confident than ours with a too-deep identification with the abstract beauty of the logical structures of computation. The insight of agile, in getting back to fundamentals of how we can work together to deliver things of human as well as commercial value, is compelling; the thrill of lean lies in the discovery that we are not alone.

[13] Kent Beck, *Software is Different.* Keynote at OT99 conference, in which all these metaphors were examined and found lacking.