

---

# THE CHALLENGES OF BRINGING LEAN TO SOFTWARE DEVELOPMENT

---

Software is not about making something, it's about making something work. Take software out of a cell phone, a car, a factory, a business, and it won't work anymore. Software is the brains behind just about everything complicated that we try to do.

In 1960, J.C.R. Licklider, the visionary behind the Internet, wrote: "The main aims [of software] are 1) to let computers facilitate formulative thinking as they now facilitate the solution of formulated problems, and 2) to enable men and computers to cooperate in making decisions and controlling complex situations without inflexible dependence on predetermined programs"<sup>1</sup> In 1962, Licklider articulated his vision of an "intergalactic network" connecting everyone in the world.<sup>2</sup> Throughout his life, Licklider could be found in a leadership role in one of the organizations that worked to convert his vision into reality: Bolt Beranek and Newman(BBN), ARPA, IBM Research, MIT. He died in 1990, just as the ideas he championed finally grew into the Internet.

If we look at the Internet through Licklider's eyes, we see the real purpose of software – it does the routine thinking for us so that we can turn our minds to more complex problems. If we look at the evolution of the Internet, we see what developing software is really all about. At its core, software development is the process of gradually finding ways to turn over more and more of what we know to computers so that we have more space left in our minds to discover ever more interesting things.

## *The Truck-Driving Problem*

Imagine that you are responsible for driving a truck across America, along highways, through cities and around detours, dealing with whatever idiosyncrasies that weather and traffic might throw at you. Now imagine that your job is not to drive the truck, but program a computer to drive the truck for you. How would you go about this task?

Many people – especially those who have never built software – would break this problem into a series of steps: first design the logic of the system; second, translate that logic into code; and finally, add the code to the truck and test that it works. But you can imagine that a problem as complex as the truck-driving problem is far too complex to solve with this approach. Instead, the problem is being tackled over time with a lot of experimentation and testing, one module at a time. Today we have laser following cruise control and tomorrow we will have lane following capability. Soon we might expect congestion detectors and GPS guidance. Someday the truck-driving problem might be solved, but by the time it is, the system will no doubt be a lot different than what we imagine today.

## *Conventional Wisdom*

Unfortunately, it has been conventional wisdom in software development to separate design from coding, coding from testing, and software from the system it supports. This gives us development

---

<sup>1</sup> Licklider, L.C.R. (1960). "Man-Computer Symbiosis." IRE Transactions on Human Factors in Electronics, v.HFE-1. p.4-11.

<sup>2</sup> This was described in a series of memos.

processes that ignore the learning loops necessary to discover the logic of a complex system, create concrete representations of the system, make sure that these work together, and over time, grow a system that becomes capable of accomplishing the complete job. In software development, we have been asked to solve too many truck-driving problems. And when it turns out that we have been handed an impossible problem, it's usually the developers – not the process or the scale of the problem – that are held responsible for the failure.

When I was an IT manager in a manufacturing plant in the early 1980's, we discovered that the conventional wisdom about the way that manufacturing should be done had become obsolete. We figured this out because our competitors in Japan started doing things that produced far better results than we were able to achieve. In response to this serious competitive threat, we threw out conventional wisdom and decided to try the counterintuitive approach being used in Japan, called "Just-in-Time." As a management team, we came to agree on a few basic principles and enlisted the wisdom of everyone in the plant to figure out how to make these principles work in practice. Indeed, it took some time and a lot of hard work to implement Just-in-Time in our plant. As part of the effort, we completely changed the way we thought about quality and used statistics to gain a deeper insight into how our equipment worked. Slowly, it seemed to us, we got better. In retrospect, the improvement was dramatic, but it certainly didn't feel like it at the time.

Today, the name "Just-in-Time" has been replaced with "Lean," but to me it remains largely about the same thing: abandoning conventional wisdom that has become obsolete and moving on to a new way of thinking. The transformation will be multifaceted, principle-based, and entail a lot of very hard work by everyone, workers and managers alike. And – probably – it will only happen if there is something that shakes people up enough to cause them to abandon conventional wisdom and look for a new way of thinking about how to do their job.

### *Analogies*

For those who look upon Lean as a set of operational practices, it doesn't make sense to apply Lean to software development. Inventory, motion, variation, overproduction – the wastes of an operational process – don't translate easily to the job of creating the brains of a system and making it smarter over time. In fact, software development has long been plagued by inappropriate comparisons with other fields. Too often, analogies are created by someone who has only a passing acquaintance with a field. For example, someone who knows little about construction might say: "Buildings are completely designed before construction starts – if it works for buildings, why can't it work for software?" I once discussed this analogy with a group of construction superintendents, who just about laughed me out of the room. They live in a world of incomplete and conflicting drawings, master schedules that bear no resemblance to reality, owners who regularly change their minds, and the constant threat of lawsuits.

Manufacturing analogies have been foisted upon software development by those who think that "cutting code" is pretty much the same thing as cutting metal. It's painful to imagine the damage that has been done to software development by the manufacturing slogan "Do it Right the First Time." I often wonder how this slogan came to mean that code should never have to be changed once it is committed to the code base. In our plant, product specs and manufacturing processes changed all the time; we just tested a product after each tiny step to be sure that no defects had appeared. Declaring that we should not have to change code once it is written is like decreeing that once we start our truck out across the country we should never have to change the planned route, no matter what detours or bad weather or flat tires we might encounter along the way.

I was only vaguely aware of the dangers of analogies when I set out to compare Lean software development to Lean manufacturing. I like using analogies – consider the truck-driving problem for example. I think of analogies as tools to help us understand and agree upon principles; but they should never be used as implementation guidelines. Copying specific practices from another field – or even from another company in our field – is a lazy way to implement change. First class execution is possible only when teams carefully think through the principles behind new ideas, understand how these principles apply to their world, and carefully adapt specific practices so that they make sense in the new context.

### *Paradigm Shift*

I didn't run into the term "waterfall" until 1999, and when I learned what it meant, I did not understand how it could possibly work. Therefore I was not surprised to discover that "waterfall" processes really don't work – what surprised me was the fact that conventional wisdom held that they *should* work. I decided that the time had come for a paradigm shift in software development, similar to the paradigm shift that manufacturing experienced during the 1980's and early 90's.

Paradigm shifts require that someone with a clear understanding of a core problem in a particular field demonstrate a new and significantly better way to address this core problem. There's no question in my mind that in manufacturing, the core problem was push scheduling – at least it certainly was in our plant. Simply implementing an effective pull system drove all sorts of other good behavior, and led eventually to significant cost reductions. We decommissioned a lot of software when we implemented our pull system, because the logic of our existing scheduling system was fundamentally flawed.

In that 1999 waterfall project, the core problem I experienced was the idea that development should be a series of handoffs: Analysis, Design, Coding, Testing, Deployment. The problem wasn't that these steps were unimportant, but that each step was supposed to be completed before the next step began – and since the next step would be done by a different group of people, the handoff had to be done through documentation. The counterintuitive concept in Lean software development is to turn this process on its side and deploy completely done software in small increments. Let's not even try to develop a system that will drive a truck, let's see if it is possible to add some intelligence to cruise control and get that working while there is still a driver behind the wheel. Of course this means we are not sure how a fully automated truck will work, when it might be ready or how much it might cost. But if we keep the ultimate goal of an automated truck in mind and design our laser following cruise control wisely, we will be a step closer to a self-driving truck.

It might seem obvious that a completely automated truck is far too ambitious an undertaking, so splitting it up into incremental steps is the best way to make progress. What is counterintuitive is that this approach should scale downward. But it does. And if you look at any really successful complex system, this is the way it was developed. Lean development requires a visionary leader and a sponsor willing to fund the vision rather than a promise that is unlikely to be kept. However, such a sponsor can be hard to find; sponsors always seem to want a "plan."

### *The "Plan"*

The core problem in software development can be boiled down to the intense pressure for a "plan" that lays out what the software will be able to do, how long it will take, and what it will cost – and the subsequent expectation that development will follow this "plan". The problem is magnified when the software is expected to be developed independently from the system for which the

software will supply the brains. Why is planning a problem? Isn't a plan essential? As Dwight Eisenhower once said: planning is indispensable, but plans are useless.

The problem with software is that the logic behind any system is inherently complex, and developing that logic independently from the system increases the complexity by at least an order of magnitude. As Fred Brooks once noted,<sup>3</sup> "The complexity of software is an essential property... Many of the classical problems of developing software products derive from this essential complexity and its nonlinear increase with size." Trying to plan a large software development effort is not much different than trying to plan the development of a software package to drive a truck across America – without access to the truck.

You might notice that the core problem of software development is beginning to bear some resemblance to the core problem of manufacturing – we suffer from push scheduling in manufacturing and push planning in software development. But the antidote is somewhat different. In manufacturing, switching from push to pull scheduling gives dramatic results. In software development, switching from forecast-driven to feedback-driven development gives equally dramatic results.

### *The Antidote*

Embedded in every development "plan" is a forecast – a guess really – of what will happen in the future. Plans that separate the logic of a system from the system itself are based on an additional layer of guessing – ahem – forecasting. If we base the development of a complex system on forecasts, we are doing forecast-driven development. If we develop and deploy small useful feature sets and get them working incrementally, then we are doing feedback-driven development. For simple systems, either approach will work. For large, complex or evolving systems, the only feasible approach is feedback-driven development.

There is a great appetite for predictability in any business that has to answer to shareholders or any organization that is under public scrutiny. This is entirely appropriate. However, the idea that predictability comes from creating a plan and then following the plan is fundamentally flawed when complex systems are involved. Following a plan will create predictability only to the extent that the plan is based on an accurate forecast of the future and a thorough understanding of the complexity of the system to be developed. Since I am a process control engineer, I would like to call to your attention something we learned a long time ago: When you really need predictability, you create a feedback system that can respond to change as it occurs, rather than a predictive system that is incapable of dealing with variation.

Many people are concerned about feedback-driven development – it seems to imply a lack of control. But when we are trying to create the brains behind a system, control is hardly what it is all about. The vision behind the Internet had nothing to do with control – and everything to do with finding ways for people to communicate and think more effectively. It took decades of detailed work engaging great minds from around the world for the Internet to become a reality, and it will continue to emerge for decades to come. This great network of software started with a vision and grew with careful guidance, detailed technical work, and extensive collaboration. If you want to understand how Lean software development works, you need look no further.

Mary Poppendieck

---

<sup>3</sup> The Mythical Man Month, 20<sup>th</sup> Anniversary Edition, by Fred P. Brooks, Jr., Addison Wesley, 1995.