

Lean Software Development: A Tutorial

Mary Poppendieck, Poppendieck.LLC

Michael A. Cusumano, Massachusetts Institute of Technology

// This tutorial describes where lean software development comes from, what it means, how it relates to well-known agile development practices, and how it is likely to evolve in the future. //



THE TERM “LEAN” was first applied publicly to a production management process and then to product development at MIT during the mid-1980s (for a history of lean in the operations management literature, see Matthias Holweg’s “The Genealogy of Lean Production”¹). John Krafcik, currently the CEO of Hyundai Motor America, was the first American engineer hired by Toyota to work in the NUMMI (New United Motor Manufacturing, Inc.) assembly plant in California, a joint venture with General Motors. He came to the MIT Sloan School of Management in 1986 to do a master’s degree

and thesis under my (Michael’s) supervision, after I had just published *The Japanese Automobile Industry* in 1985. I had reported that Toyota, using techniques that had evolved since the late 1940s, was producing automobiles with roughly half the number of labor hours as the Big Three US automakers with much faster inventory turns and higher quality. Nissan was not far behind Toyota. Encouraged by MIT’s International Motor Vehicle Program (IMVP) research director, James Womack, Krafcik launched a survey to compare auto assembly plants around the world. In 1988 companion articles in

Sloan Management Review, I summarized my findings while Krafcik attempted to generalize Toyota’s management philosophy and presented his own data showing Japanese superiority. Krafcik coined the term “lean” as a contrast to a Ford-style “buffered” mass production system.²

Toyota’s techniques dramatically reduced in-process and final inventories, expanding worker responsibilities to operate more machines, as well as enabling more assembly line workers to build in quality. A key element was that Toyota reversed the flow of information signals (then in the form of Kanban cards) controlling production operations. This change gave Toyota factories the ability to make small batches of components “just in time” (JIT) for final assembly and shipment to dealers, while eliminating most of the “just in case” in-process and final inventories. The basic philosophy was to “pull” materials and components through the production system as needed in a continuous flow, rather than “push” them through and stockpile using fully predetermined production plans. The terms “lean” and “JIT” were later popularized by MIT’s global best-seller *The Machine That Changed the World* (Rawson, 1990). The authors used the term “lean” to describe any efficient management practice that minimized waste, including in product development, where Japanese automakers had achieved shorter leader times (by about one-third) and fewer engineering hours (by about half) compared to US and European projects. (The more detailed research was done at Harvard Business School by IMVP research affiliates Kim Clark and Takahiro Fujimoto, published as *Product Development Performance* [Harvard Business School, 1991].)

Some similarities between Japanese



management and PC-style software development were becoming apparent by the mid-1990s. For example, in *Microsoft Secrets* (Free Press, 1995), we (Michael and coauthor Richard Selby) noted a similarity in the philosophy behind Microsoft’s daily builds, where engineers had to stop and fix bugs on a daily basis (we dubbed this the “synch and stabilize” process), and Toyota’s JIT production philosophy, where workers stopped assembly lines whenever they detected problems to fix them immediately. This book did not use the term “lean” for software development; we were thinking mainly about the common application of a JIT engineering and quality-control practice, as well as the reduced levels of bureaucracy and staffing in Microsoft compared to companies such as IBM. To dive deeper into product development in the auto industry, along with an MIT doctoral student (Kentaro Nobeoka) I (Michael) launched another major research effort, summarized in the book *Thinking beyond Lean* (Free Press, 1998). This book also built on a publication by James Womack and Daniel Jones, *Lean Thinking* (Simon & Schuster, 1996), which attempted to generalize lean practices and applications. *Thinking beyond Lean* focused on newer approaches to product development, emphasizing the systematic reuse of product platforms and major components, as well as other techniques, such as short, overlapping phases (concurrent engineering) to reduce calendar time and engineering hours, but it only included a brief discussion of software development.

The popularization of the term “lean” and its association with “agile” for software product development seems to have come mainly from later efforts, such as those of one of us (Mary) and Tom Poppendieck in the

TABLE 1

Process comparison of Toyota and Microsoft.

Toyota-style “lean” production (manual demand-pull with Kanban cards)	1990s Microsoft-style “agile” development (daily builds with evolving features)
JIT “small lot” production	Development by small-scale features
Minimal in-process inventories	Short cycles and milestone intervals
Geographic concentration—production	Geographic concentration—development
Production leveling	Scheduling by features and milestones
Rapid setup	Automated build tools and quick tests
Machine and line rationalization	Focus on small, multifunctional teams
Work standardization	Design, coding, and testing standards
Foolproof automation devices	Builds and continuous integration testing
Multiskilled workers	Overlapping responsibilities
Selective use of automation	Computer-aided tools, but no code generators
Continuous improvement	Postmortems, process evolution

Source: M. Cusumano, *Staying Power*, Oxford, 2010, p. 197.

book, *Lean Software Development* (Addison-Wesley, 2003). In common with the MIT and IMVP researchers, we also emphasized eliminating waste and bureaucracy in product development, encouraged learning through short cycles and frequent builds, and promoted late changes and fast iterations, with feedback pulling changes into a product, rather than requirements documents and rigid plans pushing development work forward.

Authors who have used the term “lean” with reference to software development all seem to have stressed the difference between older and more labor-intensive, bureaucratic, push-style methods, initially associated with the mainframe computer business, and newer, less bureaucratic, iterative or incremental, and flexible methods. There clearly are many common elements between Toyota-style lean production and Microsoft-style iterative

or agile development (before the Windows teams became inordinately large in the late 1990s and early 2000s), even when we look at specific practices. My (Michael’s) 2010 book summarized some of these similarities under the principle of “pull, don’t just push” (see Table 1).

Early use of the term “lean” no doubt leveraged the popularity of Japanese management techniques, but the emphasis has always been on reducing waste in terms of time and staffing, focusing on value for the customer, the product, and the enterprise, as well as stressing the benefits of a more flexible, iterative, lightweight development process. We also see this orientation in XP and Scrum.³ These approaches, referred to as iterative, incremental, or agile, encompass a set of techniques for software development that are less sequential or “waterfall-ish” as well as bureaucratic and slow, and

have become commonplace around the world owing to the benefits they offer.⁴

Lean Software Development

This leads to the more general issue of whether or not it is even appropriate to apply the principles behind lean production to product development or software engineering. In fact, we

on how to use IT to create value for the customers.... Thus lean development is not a software engineering methodology in the conventional sense. It's really a synthesis of a system of practices, principles, and philosophy for building software systems for a customer's use."

In the book, *Lean Software Development* (Addison-Wesley, 2003) we (Mary and coauthor Tom) also

process, and so on. Moreover, the value of software isn't derived solely from the development phase; design and deployment are fundamental to its value. Finally, the value of software is rarely limited to a single time-bound effort; the capability to modify a code base over time tends to be a dominant factor in its overall value.

If lean is thought of as a set of practices, it doesn't translate well from operational to development environments.

have frequently encountered negative reactions to this with comments such as, "Development is not at all like manufacturing." And, indeed, if lean is thought of as a set of practices, it doesn't translate well from operational to development environments. However, if lean is thought of as a set of principles rather than practices, then applying lean concepts to product development and software engineering makes more sense and can lead to process and quality improvements.

In fact, the idea of applying lean principles to software development is almost as old as the term "lean" itself. In the 1990s, Robert Charette used the term "lean development" to refer to a risk management strategy that brings about dynamic stability in organizations by making them more agile, resilient, and change tolerant. In 2002, Charette wrote "Challenging the Fundamental Notions of Software Development," a piece that is as wise today as it was a decade ago.⁵ In it, he provides an excellent summary of the main points of this article: "As Drucker pointed out long ago, a business's sole purpose is to create and serve the customer. In this spirit, [lean development]'s focus is not on the development process per se, but

emphasized seven principles of lean software development (which we later modified slightly, based on subsequent experience):

- optimize the whole,
- eliminate waste,
- build quality in,
- learn constantly,
- deliver fast,
- engage everyone, and
- keep getting better.

These principles are used to frame the discussion about what product development practices might be appropriate for, depending on the unique situation of each organization.

Optimize the Whole

Lean software development should be founded on a deep understanding of a job that customers would like done and how this job might be mediated by software. Discovering what customers care about and what they will value is not a simple process, especially because software rarely has value in and of itself. The value of software is delivered in the context of a larger system: an automated car, a website for buying products, an automated order fulfillment

Eliminate Waste

In lean terms, "waste" is anything that doesn't either add customer value directly or add knowledge about how to deliver that value more effectively. Some of the biggest causes of waste in software development are unnecessary features, lost knowledge, partially done work, handovers, and multitasking, not to mention the 40 to 50 percent of development time spent finding and fixing defects. Many of these wastes have their roots in the large batches of partially done work created in sequential development processes, in the boundaries between different functions, and in the delays and knowledge lost when work crosses these boundaries. When organizations look at the flow of value across the entire value stream, causes of waste are more easily exposed and addressed, much as in a JIT pull system for manufacturing.

Build Quality In

Microsoft's synch and stabilize process seemed to be a dramatic departure from the prevailing process of waiting until the end of a development cycle before integrating small units of software into a large system. But, in fact, continuously integrating small units of software into larger systems has long been held to be best practice, even though it might not have been common practice. As long ago as 1970, IBM's Harlan Mills successfully developed an approach he called "top-down programming"—a process whereby modules are integrated into the overall system as they are written, rather than

at the end of development. In his 1988 book *Software Productivity* (Dorset House), he noted, “My principal criterion for judging whether top down programming was actually used is [the] absence of any difficulty at integration.”

Learn Constantly

In the end, development is all about creating knowledge and embedding that knowledge in a product. Lean development recommends approaching this in two different ways, depending on the context.

The first is to explore multiple options for expensive-to-change decisions such as fundamental architecture, choice of language, design language for user interaction, and so on. Delay critical decisions to the last responsible moment, and then make decisions based on the best available knowledge at the time. Because multiple options have been explored, there will always be an alternative that will work, and the option that best optimizes the overall system can be chosen. This is often called a “learn first” approach.

The second way is to build a minimum set of capabilities to get started, followed by frequent delivery, while using feedback from real customer experience to make product content decisions. This continuous learning process will minimize the effort spent developing features that customers don’t find valuable.

Although the “learn first” approach might seem diametrically opposed to the “learn constantly” approach, they can be used together effectively. The key is to use options and constraints to drive critical decisions while recognizing that the content of most software systems will change constantly over time.

Deliver Fast

In many lean development environments, production releases occur frequently—weekly, daily, even continuously. Mistake-proofing mechanisms necessary for such

frequent deployment have dramatically improved quality while eliminating the large regression overhead formerly associated with releases. Of course, existing code bases can be riddled with defects, and small perturbations can have serious unintended consequences, but many organizations have discovered how to create test harnesses that can automatically detect most defects introduced by small changes. Thus as an industry, we have moved much closer to Harlan Mills’s ideal.

When software is delivered quickly, thinking about software development as a project is an inappropriate metaphor. It’s much better to think of software as a flow system where software is designed, developed, and delivered in a steady flow of small changes. This is fundamentally different from thinking about software development as a project to be completed, or even thinking about software as a series of annual or semiannual releases.

Rapid delivery should not be isolated to software development; flow should happen within the overall product development cycle, of which software is one aspect. Embedded software devel-

opment, for example, usually takes on the flow characteristics of the product it’s embedded in. A good place to look for more information on product development flow is in *Principles of Product Development Flow* by Don Reinertsen (Celeritas Publishing, 2009).

Engage Everyone

When software is thought of as something that grows continually over time, and its development is conceived of as

a flow process, a fundamentally different organizational structure is often required. Instead of placing software development in a separate department called “IT,” software development tends to be thought of as product development and located in line business units. Responsibility for discovering, creating, and delivering value falls to a team that encompasses the complete value stream. Thus a value stream team that includes software development will almost certainly include additional functions; in fact, it will probably look like a miniature version of a business. It will include people who understand customers, designers, developers, testers, operations, support, and perhaps finance.

Even when software development occurs in a separate organization, lean practices encourage teamwork among engaged people who are empowered to make decisions appropriate to their level. Although some so-called lean implementations appear to emphasize processes over people, these represent a misunderstanding of lean principles. Empowering people, encouraging teamwork, and

Even when software development occurs in a separate organization, lean practices encourage teamwork.

moving decision-making to the lowest possible level are fundamental to any lean implementation.

Keep Getting Better

In their September 1999 *Harvard Business Review* article “Decoding the DNA of the Toyota Production System,” Steven Spear and Kent Bowen point out that at Toyota, every work system is improved constantly, using the scientific method, under the

guidance of a teacher, at the lowest possible level of the corporation. Lean thinking holds that specific practices, no matter how well they seem to work in other situations, are seldom the best solution to the problem at hand. Therefore lean thinking would recommend that organizations starting with practices such as XP or Scrum (or a combination) should think of them as a starting point that will be adapted and improved over time by the people and teams doing the work.

Agile Software Development

To put the concept of lean software development in context, it's useful to point out similarities and differences with agile software development. "Agile," a common term in everyday language today, was used in the 1990s to refer to flexible production systems.^{6,7} It was then applied to software development in February 2001 when a group of like-minded software experts produced the *Manifesto for Agile Software Development*.⁸ The manifesto stated that when developing software, it's preferable to value individuals and

appropriate for individual contexts and situations that expand beyond software development.

XP

The first agile process to become popular was XP, defined in the 2000 book *Extreme Programming Explained* (Addison-Wesley) by Kent Beck. XP focuses on several technical practices, the most notable being test-driven development. TDD results in a unit test harness, which is run frequently, making it possible to detect defects almost immediately after they are injected into the code base. This approach has proven to be an effective first step in mistake-proofing a code base. Over time, methods have developed, keeping the test harness runtime to a minimum while factoring the tests so they don't grow beyond maintainable proportions.

Eventually, the concept behind TDD was expanded to include automated product specifications, which set the stage for automated regression testing. New techniques such as Framework for Integrated Testing (FIT)⁹ and its partner Fitness, acceptance test-driven development (ATDD), behavior-driven

(Prentice Hall, 2001) by Ken Schwaber and Mike Beedle. Over the next few years, Scrum became increasingly popular as a method to replace traditional project management with development iterations of one month (and later two weeks). Thus it has been an excellent way of introducing the lean concept of flow into software development. Because of Scrum's widespread popularity, it's often equated with agile. However, it doesn't claim to be a complete methodology—it lacks technical practices such as those found in XP and is generally limited to the software development portion of the value stream. Nevertheless, these early agile processes proved that incremental delivery of software was a viable approach for many domains, and that quality could be significantly improved through a disciplined approach to test-first development.

Roles

Both Scrum and XP are sets of practices aimed at optimizing the software development process as a separate activity in the value stream. They each define a role—the "customer" in XP and the "product owner" in Scrum—to own the responsibility for deciding what needs to be done. Development team members aren't generally expected to assume the responsibility for the overall success of their work; that responsibility is delegated to the customer or product owner roles. In practice, the implementation of these roles has frequently led organizations to violate the lean principle of optimizing the whole.

Lean software development, as we view the term, places software development as a step in a product value stream, regards developers as members of a larger product team, and expects all product team members to become engaged in the overall success of the product. There's no product owner or customer roles in lean development. Lean development teams are led by

Early agile processes proved that incremental delivery of software was a viable approach for many domains.

interactions over processes and tools; working software over comprehensive documentation; customer collaboration over contract negotiation; and responding to change over following a plan. These precepts were a reaction against common software development practices at the time, and they're certainly compatible with lean principles. However, we see lean principles as providing more comprehensive guidance for choosing development practices

development (BDD), and specification by example (SBE)¹⁰ emerged to tackle this highly domain-dependent problem. Automated testing frameworks are currently the most common technique used to implement the lean principle of building quality in.

Scrum

The next agile process to gain traction was Scrum, defined in the book *Agile Software Development with SCRUM*

someone in a role such as chief engineer (Toyota), program manager (Microsoft), or product champion (3M), whose roles are similar to an entrepreneur leading a startup business. Although these roles might seem similar to the customer or product owner roles in agile software development, they're quite different in practice. A chief engineer has overall responsibility for a complete product, including its success in the marketplace, and engages everyone on a multidiscipline product team in delivering that success. There's no intermediate role prioritizing work for a separate software development team.

Kanban

In his 2009 book, *Scrumban: Essays on Kanban Systems for Lean Software Development* (Modus Cooperandi Press), Corey Ladas described how to use a Kanban (card) system to both track and limit work in progress in a software development environment. This was followed in 2010 by David Anderson's book *Kanban* (Blue Hole Press, 2010), which describes how to use Kanban as the basis of an effective, flow-based software development system.

In a Kanban system, the value stream is mapped on a chart (preferably a physical chart on a wall) with columns for each step in the value stream. Kanban cards (tokens for a piece of work) are placed in the column, representing the current state of the work. When work meets specified policies for being complete, the Kanban token moves to the next column, and over time, the card moves from left to right across the board—you visually see work flow through the system. The key to a Kanban system is that work in any column (representing a step in the value stream) is limited. This means that within any value-adding activity, there's a limited amount of work; moreover, the entire system contains a limited amount of work.

Kanban systems provide a good framework for organizations getting started with lean principles. Because there are very few rules or roles, Kanban systems require thoughtful consideration and adaptation. For example, a Kanban board might start out in a software development environment, but can easily expand to include more steps

A senior manager at this company recently noted that, in his opinion, speed drives all other lean principles: "If you deliver daily, waste is exposed almost immediately; you have no choice but to build quality in; you learn quickly what customers value; everyone at every level is focused on making customers happy; problems are exposed quickly and so

Kanban systems provide a good framework for organizations getting started with lean principles.

in the value stream, such as marketing and operations. This makes Kanban a good tool for value stream teams. Kanban systems specifically focus on the flow of value; in fact, flow and bottlenecks are the main topic of daily meetings. Finally, Kanban board layouts and policies are expected to be evaluated and improved on a regular basis. For an excellent case study of the use of a Kanban system for a large government contract, see Henrik Kniberg's book *Lean from the Trenches* (Pragmatic Bookshelf, 2011).

Trends

Throughout the last decade, increasingly sophisticated tools have evolved, making the development process more mistake-proof while safely allowing the delivery of a continuous flow of small features into production. These tools were initially developed for Web-based platforms delivering software as a service.

Continuous Delivery

As early as 2004, a large Web-based enterprise (which will remain anonymous for the purposes of this article) released to production all of the software that had been worked on during the day at the end of every single day.

constant improvement is mandatory; and finally, optimizing just a part of the system simply is not an option with daily deployment."

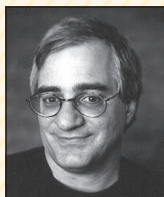
In 2011, Jez Humble and David Farley's book *Continuous Delivery* (Addison-Wesley Professional, 2010) described the technical practices that must be in place to enable a continual flow of new software released to a production environment in a safe, defect-free manner. This book, written for enterprise IT departments, lays out the tools, technologies, and organizational cooperation necessary to achieve the lean ideal of deploying software safely to production virtually immediately after it's written. Forward-looking IT departments around the world have followed this playbook and found that it takes about a year of hard work to reach a goal of weekly, daily, or even continuous delivery.

Lean Startup

Agile methods usually provide for a customer or customer proxy to direct the work of the software development team; however, they provide few mechanisms to ensure that what the customer proxy asks for will effectively address the customer problem, nor do they typically involve the development



MARY POPPENDIECK is retired from 3M and is currently president of Poppendieck.LLC. Her industry experience includes software development, supply-chain management, manufacturing operations, and new product development. Poppendieck has an MS in mathematics from the University of Maryland. Contact her at mary@poppendieck.com.



MICHAEL A. CUSUMANO is the Sloan Management Review Distinguished Professor at the Massachusetts Institute of Technology's Sloan School of Management, with a joint appointment in MIT's Engineering Systems Division. His research interests include technology management and strategy, especially in the software business. Cusumano has a PhD in Japanese studies from Harvard University. Contact him at cusumano@mit.edu.

team in assuring that the product is a market success.


In 2011, Eric Reis addressed this problem in the book *Lean Startup* (Crown Business, 2011) where he advocates that companies start with a business plan and test the impact of features on the key assumptions that form the basis of that plan. In this approach, the software development team becomes involved in setting up the feedback loops, such as split tests, necessary to determine the impact of features. Thus, software engineers are typically involved in the process of validating the value of new features to customers while making adjustments based on this feedback. Obtaining feature-by-feature feedback is an effective way for companies targeting a large customer base to validate the value of various approaches.

Design Thinking

Over the past decade, the epicenter of software value creation has changed. It used to be that orchestrating transactions and controlling equipment were the primary purposes of software. Today, new purposes have emerged, such

as providing platforms for two-sided markets and creating engaging experiences. The most rapidly growing software companies provide ecosystems that attract traffic through their ability to understand and address an important customer need that is not being adequately served. Increasingly, the design of the customer experience is a foundational element of such an ecosystem. At the same time, these systems tend to use a continuous delivery approach rather than a project approach, so the system architecture must be designed from the beginning to support dynamic updating and continuous change.

Agile development methods have generally expected system architecture and interaction design to occur outside the development team, or to occur in very small increments within the team. Because of this, agile practices often prove to be insufficient in addressing issues of solution design, user interaction design, and high-level system architecture.

Increasingly, agile development practices are being thought of as good ways to organize software development, but insufficient ways to address design. Because design is fundamentally iterative and development is fundamentally iterative, the two disciplines suffer if they are not carefully integrated with each other. Because lean development lays out a set of principles that demand a whole-product, complete life-cycle, cross-functional approach, it's the more likely candidate to guide the combination of design, development, deployment, and validation into a single feedback loop focused on the discovery and delivery of value. 

References

1. M. Holweg, "The Genealogy of Lean Production," *J. Operations Management*, vol. 25, no. 2, 2007, pp. 420–437.
2. J. Krafcik, "Triumph of the Lean Production System," *MIT Sloan Management Rev.*, vol. 30, no. 1, 1988, pp. 41–52.
3. M. Cusumano, "Extreme Programming Compared with Microsoft-Style Iterative Development," *Comm. ACM*, vol. 50, no. 10, 2007, pp. 15–18.
4. M. Cusumano et al., "Software Development Worldwide: The State of the Practice," *IEEE Software*, vol. 20, no. 6, 2003, pp. 28–34.
5. R.N. Charette, "Challenging the Fundamental Notions of Software Development," white paper, IT Metrics and Productivity Inst., 2003; www.itmpi.org/assets/base/images/itmpi/privaterooms/robertcharette/ChallengingtheFundamentalNotions.pdf.
6. P.T. Kidd, *Agile Manufacturing: Forging New Frontiers*, Addison-Wesley, 2004.
7. R. DeVor, R. Graves, and J. Mills, "Agile Manufacturing Research: Accomplishments and Opportunities," *Inst. Industrial Engineers Trans.*, vol. 29, 1997, pp. 813–823.
8. K. Beck et al., *Manifesto for Agile Software Development*, 2001; <http://agilemanifesto.org>.
9. R. Mugridge and W. Cunningham, *FIT for Developing Software*, Addison-Wesley, 2005.
10. G. Adzic, *Specification by Example*, Manning Publications, 2011.



Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.