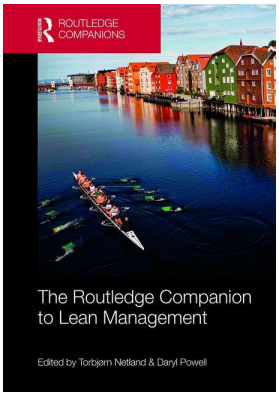

The Routledge Companion to Lean Management

Torbjørn H. Netland, Daryl J. Powell

Lean Software Development

Publication details
https://test.routledgehandbooks.com/doi/10.4324/9781315686899.ch34
Mary Poppendieck
Published online on: 28 Dec 2016

34
LEAN SOFTWARE DEVELOPMENT

Mary Poppendieck

Introduction

We were in a conference room near the Waterfront in Cape Town. "I just lost a crown from one of my teeth," my husband, Tom, declared just before I was scheduled to open the conference. Someone at our table responded, "You're lucky, Cape Town has some of the best dentists in the world." It didn't feel very lucky: Cape Town was the first stop on a 10-week trip to Africa, Europe, and Australia.

The situation was eerily familiar. A year earlier a chip had cracked off my tooth as I ate a pizza in Lima, the first stop of a 10-week trip to South America. I ate gingerly during the rest of the trip, worried that the tooth would crack further. Luckily, I made it back home with no pain and little additional damage. Once there, it took three days to get a dentist appointment. The dentist made an impression of the gap in my tooth and fashioned a temporary crown. "This will have to last for a week or two," she said. "If it falls out, just stick it back in and be more careful what you eat." Luckily the temporary crown held, and 10 days later a permanent crown arrived from the lab. Two weeks after we arrived home, my tooth was fixed.

We were scheduled to be in Cape Town for only two days. How was Tom going to get a crown replaced in two days? A small committee formed. Someone did a phone search; apparently the Waterfront was a good place to find dentists. A call was made. "You can go right now—the dental office is nearby. Do you want someone to walk you over?" As Tom headed out the door with an escort, I got ready for my presentation. Halfway through the talk, I saw Tom return and signal that all was well.

"I lost a part of my tooth, not just the crown," Tom told me after the talk. "I'm supposed to return at 3.30 this afternoon; I should have a new crown by the end of the day." The dentist had a mini-lab in his office. Instead of making a temporary crown, he used a camera to take images of the broken tooth and adjacent teeth. The results were combined into a 3D model of the crown to which the dentist made a few adjustments. He then selected a ceramic blank that matched the color of Tom's teeth and put it in a milling machine. With the push of a button, instructions to make the crown were loaded into the machine. Cutters whirled and water squirted to keep the ceramic cool. Ten minutes later the crown was ready to cement in place. Ninety minutes after he arrived that afternoon and eight hours after the incident, Tom walked out of the dental office with a new permanent crown. It cost approximately the same amount as my crown had cost a year earlier.

Lean is About Flow Efficiency

The book *This is Lean* (Modig and Åhlström, 2012) describes "lean" as a relentless focus on efficiency—but not the kind of efficiency that cuts staff and money, nor the kind of efficiency that strives to keep every resource busy all of the time. In fact, a focus on resource efficiency will almost always destroy overall efficiency, the authors contend, because fully utilized machines (and people) create huge traffic jams. These jams end up creating a lot of extra work. Instead, Modig and Åhlström demonstrate that lean is about *flow efficiency*—that is, the efficiency with which a unit of work (a flow unit) moves through the system.

Consider our dental experience. It took two weeks for me get a new crown, but in truth, only an hour-and-a-half of that time was needed to actually fix the tooth; the rest of the time was mostly spent waiting. My flow efficiency was $1.5 \div 336$ (two weeks) = 0.45 percent. On the other hand, Tom's tooth was replaced in eight hours—42 times faster—giving him a flow efficiency of $1.5 \div 8 = 18.75$ percent.

In my case, the dental system was focused on the efficiency of the lab's milling machine—no doubt an expensive piece of equipment. Add up all of the extra costs: a cast of the crown for the lab, a temporary crown for me, two separate hour-long sessions with the dentist, plus all of the associated logistics—scheduling, shipping, tracking, etc. In Tom's case, the dental system was focused on the speed with which it could fix his tooth—which was good for us, because a long wait for a crown was not an option. True, the milling machine in the dentist's office sits idle much of each day. (The dentist said he has to replace two crowns a day to make it economically feasible.) However, when you add up the waste of temporary crowns, the piles of casts waiting for a milling machine, and the significant cost of recovering from a mistake—an idle milling machine makes a lot of sense.

What does flow efficiency really mean? Assume you have a camera and efficiency means keeping the camera busy—always taking a picture of some value-adding action. Where do you aim your camera? In the case of resource efficiency, the camera is aimed at the resource—the milling machine—and keeping it busy is of the utmost importance. In the case of flow efficiency, the camera is on the flow unit—Tom—and work on replacing his crown is what counts. The fundamental mental shift that lean requires is this: flow efficiency trumps resource efficiency almost all of the time.

What is Lean Software Development?

To understand *lean software development*, we first need to understand lean product development. During the 1980s Japanese cars were capturing market share at a rate that alarmed US automakers. In Boston, both MIT and Harvard Business School responded by launching extensive studies on the automotive industry. In 1990 the MIT research effort resulted in the now classic book *The Machine that Changed the World: The Story of Lean Production* (Womack et al., 1990), which popularized the term "lean." A year later, Harvard Business School published *Product Development Performance* (Clark and Fujimoto, 1991), and the popular book *Developing Products in Half the Time* (Smith and Reinertsen, 1991) was released. These two 1991 books are foundational references on what came to be called "lean product development," although the term "lean" would not be associated with product development for another decade.

Clark and Fujimoto documented the fact that US and European volume automotive producers took three times as many engineering hours and 50 percent more time to develop a car compared with Japanese automakers, yet the Japanese cars had substantially higher quality and cost less to manufacture. Clearly the Japanese product development process produced better cars

faster and at lower cost than typical Western development practices of the time. Clark and Fujimoto noted that the distinguishing features of Japanese product development paralleled features found in Japanese automotive production. For example, Japanese product development focused on flow efficiency, reducing information inventory, and learning based on early and frequent feedback from downstream processes. By contrast, product development in Western countries focused on resource efficiency, completing each phase of development before starting the next, and following the original plan with as little variation as possible.

In 1991, the University of Michigan began its Japan Technology Management Program. Over the next several years, faculty and associate members included Jeffrey Liker, Allen Ward, Durward Sobek, John Shook, and Mike Rother. This group has published numerous books and articles on lean thinking, lean manufacturing, and lean product development, including *The Toyota Product Development System* (Morgan and Liker, 2006) and *Lean Product and Process Development* (Ward, 2007). The second book summarizes the essence of lean product development this way:

1 Understand that knowledge creation is the essential work of product development.
2 Charter a team of responsible experts led by an entrepreneurial system designer.
3 Manage product development using the principles of cadence, flow, and pull.

It is important to recognize that even though lean product development is based on the same principles as lean production, the practices surrounding development are not the same as those considered useful in production. In fact, transferring lean practices from manufacturing to development has led to some disastrous results. For example, lean production emphasizes reducing variation—exactly the wrong thing to do in product development. The Western practice of following a plan and measuring variance from a plan is often justified by the slogan "Do it right the first time." Unfortunately, this approach does not allow for learning; it confines designs to those conceived when the least amount of knowledge is available. A fundamental practice in lean product development is to create variation (not avoid it) in order to explore the impact of multiple approaches. This is called set-based engineering (see Chapter 6 in this book, "Lean Product and Process Development" by Rossi, Morgan and Shook).

The critical thing to keep in mind is that knowledge creation is the essential work of product development. While lean production practices support learning about and improving the manufacturing process, their goal is to minimize variation in the product. This is not appropriate for product development, where variation is an essential element of the learning cycles that are the foundation of good product engineering. Therefore, instead of copying lean manufacturing practices, lean product development practices must evolve from a deep understanding of fundamental lean principles adapted to a development environment.

Lean Software Development is a Subset of Lean Product Development

In 1975, computers were large, expensive, and rare. Software for these large machines was developed in the IT departments of large companies and dealt largely with the logistics of running the company—payroll, order processing, inventory management, etc. However, as mainframes morphed into minicomputers, personal computers, and microprocessors, it became practical to enhance products and services with software. Then the internet began to invade the world, and it eventually became the delivery mechanism for a large fraction of the software being developed today. As software moved from supporting business process to enabling smart products and becoming the essence services, software engineers moved from IT departments to line organizations where they joined product teams.

Today, most software development is not a stand-alone process, but rather a part of developing products or services. Thus lean software development might be considered a subset of lean product development; certainly the principles that underpin lean product development are the same principles that form the basis of lean software development.

Agile and Lean Software Development: 2000–2010

It is hard to believe these days, but in the mid-1990s developing software was a slow and painful process found in the IT departments of large corporations. As the role of software expanded and software engineers moved into line organizations, reaction against the old methods grew. In 1999, Kent Beck proposed a radically new approach to software development in the book *Extreme Programming Explained* (Beck, 2000). In 2001 the *Agile Manifesto* (Beck et al., 2001) gave this new approach a name—"agile."

In 2003, the book *Lean Software Development* (Poppendieck and Poppendieck, 2003) merged lean manufacturing principles with agile practices and the latest product development thinking, particularly from the book *Managing the Design Factory* (Reinertsen, 1997). Lean software development was presented as a set of principles that form a theoretical framework for developing and evolving agile practices:

1 eliminate waste,
2 amplify learning,
3 decide as late as possible,
4 deliver as fast as possible,
5 empower the team,
6 build quality in, and
7 see the whole.

Although the principles of lean software development are consistent with lean manufacturing and (especially) lean product development, the specific practices that emerged were tailored to a software environment and aimed at the flaws in the prevailing software development methodologies. One of the biggest flaws at the time was the practice of moving software sequentially through the typical stages of design, development, test, and deployment—with handovers of large inventories of information accumulating at each stage. This practice left testing and integration at the end of the development chain, so defects went undetected for weeks or months before they were discovered. Typical sequential processes reserved a third of a release cycle for testing, integration, and defect removal. The idea that it was possible to "build quality in" was not considered a practical concept for software.

To counter sequential processes and the long integration and defect removal phase, agile software development practices focused on fast feedback cycles in these areas:

1 *Test-driven development*: Start by writing tests (think of them as executable specifications) and then write the code to pass the tests. Put the tests into a test harness for ongoing code verification.
2 *Continuous integration*: Integrate small increments of code changes into the code base frequently—multiple times a day—and run the test harness to verify that the changes have not introduced errors.
3 *Iterations*: Develop working software in iterations of two to four weeks; review the software at the end of each iteration and make appropriate adjustments.

4 *Cross-functional teams*: Development teams should include customer proxies and testers as well as developers to minimize handovers.

During its first decade, agile development moved from a radical idea to a mainstream practice. This was aided by the widespread adoption of Scrum, an agile methodology which institutionalized the third and fourth practices listed above, but unfortunately omitted the first two practices.

The Difference between Lean and Agile Software Development

When it replaced sequential development practices typical at the time, agile software development improved the software development process in most cases—in IT departments as well as product development organizations. However, the expected organizational benefits of agile often failed to materialize because agile focused on optimizing software development, which frequently was not the system constraint. Lean software development differed from agile in that it worked to optimize flow efficiency across the entire value stream "from concept to cash." (Note the subtitle of the book *Implementing Lean Software Development: From Concept to Cash* by Poppendieck and Poppendieck, 2006.) The end-to-end view was consistent with the work of Taiichi Ohno, who said:

> All we are doing is looking at the timeline, from the moment the customer gives us an order to the point when we collect the cash. And we are reducing that time line by removing the non-value-added wastes.
>
> *(Ohno, 1988. p ix)*

Lean software development came to focus on these areas:

1 *Build the right thing*: Understand and deliver real value to real customers.
2 *Build it fast*: Dramatically reduce the lead time from customer need to delivered solution.
3 *Build the thing right*: Guarantee quality and speed with automated testing, integration, and deployment.
4 *Learn through feedback*: Evolve the product design based on early and frequent end-to-end feedback.

Let's take a look at each principle in more detail:

1 *Understand and deliver real value to real customers*: A software development team working with a single customer proxy has one view of the customer interest, and often that view is not informed by technical experience or feedback from downstream processes (such as operations). A product team focused on solving real customer problems will continually integrate the knowledge of diverse team members, both upstream and downstream, to make sure the customer perspective is truly understood and effectively addressed. Clark and Fujimoto (1991) call this "integrated problem solving" and consider it an essential element of lean product development.
2 *Dramatically reduce the lead time from customer need to delivered solution*: A focus on flow efficiency is the secret ingredient of lean software development. How long does it take for a team to deploy into production a single small change that solves a customer problem? Typically, it can take weeks or months—even when the actual work involved consumes only an hour.

Why? Because subtle dependencies among various areas of the code make it probable that a small change will break other areas of the code. Therefore, it is necessary to deploy large batches of code as a package after extensive (usually manual) testing. In many ways the decade of 2000–2010 was dedicated to finding ways to break dependencies, automate the provisioning and testing processes, and thus allow rapid independent deployment of small batches of code.

3 *Guarantee quality and speed with automated testing, integration, and deployment*: It was exciting to watch the expansion of test-driven development and continuous integration during the decade of 2000–2010. First these two critical practices were applied at the team level—developers wrote unit tests (which were actually technical specifications) and integrated them immediately into their branch of the code. Test-driven development expanded to writing executable product specifications in an incremental manner, which moved testers to the front of the process. This proved more difficult than automated unit testing, and precipitated a shift toward testing modules and their interactions rather than end-to-end testing. Once the product behavior could be tested automatically, code could be integrated into the overall system much more frequently during the development process—preferably daily—so software engineers could get rapid feedback on their work.

   Next, the operations people got involved and automated the provisioning of environments for development, testing, and deployment. Finally, teams (which now included operations) could automate the entire specification, development, test, and deployment processes—creating an automated deployment pipeline. There was initial fear that more rapid deployment would cause more frequent failure, but exactly the opposite happened. Automated testing and frequent deployment of small changes meant that risk was limited. When errors did occur, detection and recovery was much faster and easier, and the team became a lot better at it. Far from increasing risk, it is now known that deploying code frequently in small batches is the best way to reduce risk and increase the stability of large complex code bases.

4 *Evolve the product design based on early and frequent end-to-end feedback*: To cap these remarkable advance, once product teams could deploy multiple times per day, they began to close the loop with customers. Through canary releases, A/B testing, and other techniques, product teams learned from real customers which product ideas worked and how to fine-tune their offerings for better business results.

When these four principles guided software development in product organizations, significant business-wide benefits were achieved. However, IT departments found it difficult to adopt the principles because they required changes that lay beyond span of control of most IT organizations.

Lean Software Development: 2010–2015

In 2010, two significant books about lean software development were published. David Anderson's book *Kanban* (Anderson, 2010) presented a powerful visual method for managing and limiting work-in-process (WIP). Just at the time when two-week iterations began to feel slow, *kanban* gave teams a way to increase flow efficiency while providing situational awareness across the value stream. Jez Humble and Dave Farley's book *Continuous Delivery* (Humble and Farley, 2010) walked readers through the steps necessary to achieve automated testing, integration, and deployment, making daily deployment practical for many organizations. A year later, Erik Ries' book *The Lean Startup* (Ries, 2011) showed how to use the rapid feedback loop created by continuous delivery to run experiments with real

customers and confirm the validity of product ideas before incurring the expense of detailed implementation.

Over the next few years, the ideas in these books became mainstream and the limitations of agile software development (software-only perspective and iteration-based delivery) were gradually expanded to include a wider part of the value stream and a more rapid flow. A grassroots movement called DevOps worked to make automated provision-code-build-test-deployment pipelines practical. Cloud computing arrived, providing easy and automated provisioning of environments. Cloud elements (virtual machines, containers), services (storage, analysis, etc.), and architectures (microservices) made it possible for small services and applications to be rapidly and independently deployed. Improved testing techniques (simulations, contract assertions) have made error-free deployments the norm.

The State of Lean Software Development in 2015

Today's successful internet companies have learned how to optimize software development over the entire value stream. They create full stack teams that are expected to understand the consumer problem, deal effectively with tough engineering issues, try multiple solutions until the data shows which one works best, and maintain responsibility for improving the solution over time. Large companies with legacy systems have begun to take notice, but they struggle with moving from where they are to the world of thriving internet companies.

Lean principles are a big help for organizations that want to move from old development techniques to modern software approaches. For example, Calçado (2015) shows how classic lean tools—value stream mapping and problem solving with five whys—were used to increase flow efficiency at Soundcloud, leading over time to a microservices architecture. In fact, focusing on flow efficiency is an excellent way for an organization to discover the most effective path to a modern technology stack and development approach.

For traditional software development, flow efficiency is typically lower than 10 percent; agile practices usually bring it up to 30 or 40 percent. Yet in thriving internet companies, flow efficiency approaches 70 percent and is often quite a bit higher. Low flow efficiencies are caused by friction—in the form of batching, queueing, handovers, and delayed discovery of defects, as well as misunderstanding of consumer problems and changes in those problems during long resolution times. Improving flow efficiency involves identifying and removing the biggest sources of friction from the development process.

Modern software development practices—the ones used by successful internet companies—address the friction in software development in a very particular way. The companies start by looking for the root causes of friction, which usually turn out to be 1) misunderstanding of the customer problem, 2) dependencies in the code base, and 3) information and time lost during handovers and multitasking. Therefore, they focus on three areas:

1 understanding the consumer journey,
2 architecture and automation to expose and reduce dependencies, and
3 team structures and responsibilities.

Today, lean development in software usually focuses on these three areas as the primary way to increase efficiency, assure quality, and improve responsiveness in software-intensive systems.

1 *Understand the customer journey*: Software-intensive products create a two-way path between companies and their consumers. A wealth of data exists about how products are used, how

consumers react to a product's capabilities, opportunities to improve the product, and so on. Gathering this data and analyzing it has become an essential capability for companies far beyond the internet world: car manufacturers, mining equipment companies, retail stores, and many others gather and analyze "Big Data" to gain insights into consumer behavior. The ability of companies to understand their consumers through data has changed the way products are developed (Porter and Heppelmann, 2015). No longer do product managers (or representatives from "the business") develop a roadmap and give a prioritized list of desired features to an engineering team. Instead, data scientists work with product teams to identify themes to be explored. Then the product teams identify consumer problems surrounding the theme and experiment with a range of solutions. Using rapid deployment and feedback capabilities, the product team continually enhances the product, measuring its success by business improvements not feature completion.

2 *Architecture and automation*: Many internet companies, including Amazon, Netflix, eBay, realestate.com.au, Forward, Twitter, PayPal, Gilt, Bluemix, Soundcloud, The Guardian, and even the UK Government Digital Service have evolved from monolithic architectures to microservices. They found that certain areas of their offerings need constant updating to deal with a large influx of customers or rapid changes in the marketplace. To meet this need, relatively small services are assigned to small teams which then split their services off from the main code base in such a way that each service can be deployed independently. A service team is responsible for changing and deploying the service as often as necessary (usually very frequently), while insuring that the changes do not break any upstream or downstream services. This assurance is provided by sophisticated automated testing techniques as well as automated incremental deployment.

   Other internet companies, including Google and Facebook, have maintained existing architectures but developed sophisticated deployment pipelines that automatically send each small code change through a series of automated tests with automatic error handling. The deployment pipeline culminates in safe deployments which occur at very frequent intervals; the more frequent the deployment, the easier it is to isolate problems and determine their cause. In addition, these automation tools often contain dependency maps so that feedback on failures can be sent directly to the responsible engineers and offending code can be automatically reverted (taken out of the pipeline in a safe manner).

   These architectural structures and automation tools are a key element in a development approach that uses Big Data combined with extremely rapid feedback to improve the consumer journey and solve consumer problems. They are most commonly found in internet companies, but are being used in many others, including organizations that develop embedded software. (See the case study below.)

3 *Team structures and responsibilities*: When consumer empathy, data analytics, and very rapid feedback are combined, there is one more point of friction that can easily reduce flow efficiency. If an organization has not delegated responsibility for product decisions to the team involved in the rapid feedback loop, the benefits of this approach are lost. In order for such feedback loops to work, teams with a full stack of capabilities must be given responsibility to make decisions and implement immediate changes based on the data they collect. Typically, such teams include people with product, design, data, technology, quality, and operations backgrounds. They are responsible for an improving set of business metrics rather than delivering a set of features. An example of this would be the UK Government Digital Service (GDS), where teams are responsible for delivering improvements in four key areas: cost per transaction, user satisfaction, transaction completion rate, and digital take-up.

It is interesting to note that UK laws make it difficult to base contracts on such metrics, so GDS staffs internal teams with designers and software engineers and makes them responsible for the metrics. Following this logic to its conclusion, the typical approach of IT departments—contracting with their business colleagues to deliver a pre-specified set of features—is incompatible with full stack teams responsible for business metrics. In fact, it is rare to find separate IT departments in companies founded after the mid-1990s (which includes virtually all internet companies). Instead, these newer companies place their software engineers in line organizations, reducing the friction of handovers between departments.

In older organizations, IT departments often find it difficult to adopt modern software development approaches. This is because they have inherited monolithic code bases intertwined with deep dependencies that introduce devious errors and thwart independent deployment of small changes. One major source of friction is the corporate database, once considered essential as the single source of truth about the business, but now under attack as a massive dependency generator. Another source of friction is outsourced applications, where even small changes are difficult and knowledge of how to make them no longer resides in the company. Perhaps the biggest source of friction in IT departments is the distance between their technical people and the company's customers. Since most IT departments view their colleagues in line businesses as their customers, the technical people in IT lack a direct line of sight to the real customers of the company. Therefore, insightful trade-offs and innovative solutions struggle to emerge.

The Future of Lean Software Development

The worldwide software engineering community has developed a culture of sharing innovative ideas, in stark contrast to the more common practice of keeping intellectual property and internally developed tools proprietary. The rapid growth of large, reliable, secure software systems can be directly linked to the fact that software engineers routinely contribute to and build upon the work of their worldwide colleagues through open source projects and repositories like GitHub. This reflects the long-standing practices of the academic world but is strikingly unique in the commercial world. Due to this intense industry-wide knowledge sharing, methods and tools for building highly reliable complex software systems have advanced extraordinarily quickly and are widely available.

As long as the software community continues to leverage its knowledge-sharing culture it will continue to grow rapidly, because sophisticated solutions to seemingly intractable problems eventually emerge when many minds are focused on the problem. The companies that will benefit the most from these advances are the ones that not only track new techniques as they are being developed, but also contribute their own ideas to the knowledge pool.

As micro-structured architectures and automated deployment pipelines become common, more companies will adopt these practices, some earlier and some later, depending on their competitive situation. The most successful software companies will continue to focus like a laser on delighting customers, improving the flow of value, and reducing risks. They will develop (and release as open source) an increasingly sophisticated set of tools that make software development easier, faster, and more robust. Thus, a decade from now there will be significant improvements in the way software is developed and deployed. The lean principles of understanding value, increasing flow efficiency, eliminating errors, and learning through feedback will continue to drive the evolution, but the term "lean" will disappear as it becomes "the way things are done."

Case Study: Lean Software Development in Hewlett Packard LaserJet Firmware

The HP LaserJet firmware department had been the bottleneck of the LaserJet product line for a couple of decades, but by 2008 the situation had turned desperate. Software was increasingly important for differentiating the printer line, but the firmware department simply could not keep up with the demand for more features. Department leaders tried to spend their way out of the problem, but more than doubling the number of engineers did little to help. So they decided to engineer a solution to the problem by reengineering the development process.

The starting point was to quantify exactly where all the engineers' time was going. Fully half of the time went to updating existing LaserJet printers or porting code between different branches that supported different versions of the product. A quarter of the time went to manual builds and manual testing, yet despite this investment developers had to wait for days or weeks after they made a change to find out if it worked. Another 20 percent of the time went to planning how to use the 5 percent of time that was left to do any new work. The reengineered process would have to radically reduce the effort needed to maintain existing firmware, while seriously streamlining the build and test process. The planning process could also use some rethinking.

It is not unusual to see a technical group use the fact that they inherited a messy legacy code base as an excuse to avoid change. Not in this case. As impossible as it seemed, a new architecture was proposed and implemented that allowed all printers—past, present, and even future—to operate off the same code branch, determining printer-specific capabilities dynamically instead of having them embedded in the firmware. Of course, this required a massive change, but the department tackled one monthly goal after another and gradually implemented the new architecture. However, changing the architecture would not solve the problem if the build and test process remained slow and cumbersome. The engineers methodically implemented techniques to streamline that process. In the end, a full regression test—which used to take six weeks—was routinely run overnight. Yes, this involved a large amount of hardware, simulation, and emulation, and yes, it was expensive. However, it paid for itself many times over.

During the recession of 2008, the firmware department was required to return to its previous staffing levels. Despite a 50 percent headcount reduction, there was a 70 percent reduction in cost per printer program once the new architecture and automated provisioning system were in place in 2011. At that point there was a single code branch and 20 percent of engineering time was spent maintaining the branch and supporting existing products. Thirty percent of engineering time was spent on the continuous delivery infrastructure, including build and test automation. Wasted planning time was reclaimed by delaying speculative decisions and making choices based on short feedback loops. Moreover, there was something to plan for, because over 40 percent of the engineering time was available for innovation. This multi-year transition was neither easy nor cheap, but it absolutely was worth the effort. (For more details, see Gruver et al., 2013.)

A more recent case study of how the bookmaker and software company Paddy Power moved to continuous delivery can be found in Chen (2015). In this case study the benefits of continuous delivery are listed: improved customer satisfaction, accelerated time to market, building the right product, improved product quality, reliable releases, and improved productivity and efficiency. There is really no downside to continuous delivery. Of course, it is a challenging engineering problem, which can require significant architectural modifications to existing code bases as well as sophisticated pipeline automation.

However, technically, continuous delivery is no more difficult than other problems software engineers struggle with every day. The real stumbling block is the change in organizational structure and mindset required to achieve serious improvements in flow efficiency.

References

Anderson, D. (2010). *Kanban*, Sequim, WA, Blue Hole Press.

Beck, K. (2000). *Extreme Programming Explained*, Reading, MA, Addison-Wesley.

Beck, K., Beedle, M., van Bennekum, A. *et al.* (2001). *Manifesto for Agile Software Development*. Available at: http://agilemanifesto.org/ (accessed July 2016).

Calçado, P. (2015). How we ended up with microservices. Available at: http://philcalcado.com/2015/09/08/how_we_ended_up_with_microservices.html (accessed July 2016).

Chen, L. (2015). Continuous delivery: Huge benefits but challenges too. *IEEE Software*, *32*(2), 50–54.

Clark, K. B. and Fujimoto, T. (1991). *Product Development Performance*, Boston, MA, Harvard Business School Press.

Gruver, G., Young, M. and Fulghum, P. (2013). *A Practical Approach to Large-Scale Agile Development*, Harlow, Pearson Education.

Humble, J. and Farley, D. (2010). *Continuous Delivery*, Reading, MA, Addison-Wesley Professional.

Modig, N. and Åhlström, P. (2012). *This is Lean: Resolving the Efficiency Paradox*, Stockholm, Rheologica Publishing.

Morgan, J. M. and Liker, J. K. (2006). *The Toyota Product Development System*, New York, Productivity Press.

Ohno, T. (1988). *Toyota Production System: Beyond Large-Scale Production*, New York, Productivity Press. Published in Japanese in 1978.

Poppendieck, M. and Poppendieck, T. (2003). *Lean Software Development*, Reading, MA, Addison-Wesley.

Poppendieck, M. and Poppendieck, T. (2006). *Implementing Lean Software Development*, Reading, MA, Addison-Wesley.

Porter, M. E. and Heppelmann, J. E. (2015). How smart, connected products are transforming companies, *Harvard Business Review*, *93*(10), 97–112.

Reinertsen, D. G. (1997). *Managing the Design Factory*, New York, The Free Press.

Ries, E. (2011). *The Lean Startup*, New York, Crown Business.

Smith, P. G. and Reinertsen, D. G. (1991). *Developing Products in Half the Time*, New York, Van Nostrand Reinhold/Wiley.

Ward, A. (2007). *Lean Product and Process Development*, Cambridge, MA, Lean Enterprise Institute.

Womack, J. P., Jones, D. T. and Roos, D. (1990). *The Machine that Changed the World: The Story of Lean Production*, New York, Rawson & Associates.