Implementing Lean Software Development – From Concept To Cash

Mary and Tom Poppendieck; Addison-Wesley, 2007, 276 pages
ISBN 0-321-43738-1

Book report
Jim McDonough, August 30, 2009

Review

This book, the Poppendiecks' second on the subject of lean, is a sequel to their 2003 publication Lean Software Development – An Agile Toolkit, and takes the reader to the next level of understanding for using lean principles in software development. It begins by reviewing the history of lean and the seven principles of lean software development presented in the first book. It then presents new perspectives on the topics of value, waste, speed, people, knowledge, quality, partners and the journey ahead, each accompanied by anecdotes and accounts of the rewards and successes experienced by organizations adopting a lean approach and of the difficulties and problems experienced by those failing to embrace change. As they state in the preface:

> "We assume the reader is convinced that lean software development is a good idea, and focus on the essential elements of a successful implementation. We look at key aspects of implementation and discuss what is important, what isn't and why. Our objective is to help organizations get started down the path toward more effective software development." (p. xxiv)

The first book describes the Seven Lean Principles as shown below in the left column. This second book presents them in chapter two as the Seven Principles of Lean Software Development, shown in the right column:

1. Eliminate waste
2. Amplify learning
3. Decide as late as possible
4. Deliver as fast as possible
5. Empower the team
6. Build integrity in
7. See the whole

1. Eliminate waste
2. Build quality in
3. Create knowledge
4. Defer commitment
5. Deliver fast
6. Respect people
7. Optimize the whole

Other than using some alternative descriptions and moving principle 6 from the first book to become principle 2 of the second book, they essentially remain the same principles.

The book contains two Forewords, one written by Jeff Sutherland, the originator of the Scrum Development process, and one by Kent Beck of Three Rivers Institute. Jeff Sutherland contributes this:

> "To create the lean 'secret sauce' for your company, we have found you must systematically implement this accumulated lean wisdom. In short, you must deeply understand the Japanese concepts of Muri, Mura and Muda. Mary and Tom unfold them as the seven lean principles and the seven wastes of software development to help you easily understand how they work and know what to do.
>
> "Muri relates to properly loading a system and Mura relates to never stressing a person, system, or process. Yet many managers want to load developers at 110 percent. They desperately want to create a greater sense of 'urgency' so developers will 'work harder'. They want to micromanage teams, which stifles self-organization. These ill-conceived notions often introduce wait time, churn, death marches, burnout and failed projects.

> "When I ask technical managers whether they load the CPU on their laptop to 110 percent they laugh and say, 'Of course not. My computer would stop running!' Yet by overloading teams, projects are often late, software is brittle and hard to maintain, and things gradually get worse, not better. One needs to understand that Toyota and Scrum use a pull system that avoids stress (Mura) and eliminates bottlenecks (Muri). The developers take what they need off the Product Backlog just in time. They choose only Product Backlog that is ready to take into a Sprint, and they never take more than they can get done in a Sprint. They go faster by surfacing impediments and working with management to eliminate waste (Muda). By removing waste, they have less work to do. Productivity goes up, and they have time to deliver quality software and focus on exactly what the customers need." – Jeff Sutherland (p. xix)

1. History

I remember about 30 years ago hearing an expert on productivity proclaim that the state of the art in manufacturing at that time was attributable to these 3 significant industrial advances:

1. Eli Whitney's concept of interchangeable parts
2. Henry Ford's concept of the assembly line
3. W. Edwards Deming's concept of using statistics to achieve high quality

This chapter begins with the subchapter titled Interchangeable Parts, the concept of which was first propounded in 1785 in Paris, France by Honoré Blanc and, at the behest of Thomas Jefferson, undertaken in the United States by Eli Whitney. The second subchapter, Interchangeable People, describes the quest of Henry Ford and the assembly line. I had expected one of the subsequent subchapters to be on Deming, but instead presents the Toyota Production System, touching on the contributions of Taiichi Ohno and Shigeo Shingo of Just-In-Time flow, Autonomation, Nonstock production, Zero inspection and Lean Manufacturing.[1]

2. Principles

The Seven Principles of Lean Software Development are presented here, each one explained and accompanied by "a prevailing myth, which makes the principle counterintuitive for those who believe the myth". (p. 23):

Principle	**Prevailing myth**
1. Eliminate waste	Early specification reduces waste
2. Build quality in	The job of testing is to find defects
3. Create knowledge	Predictions create predictability
4. Defer commitment	Planning is commitment
5. Deliver fast	Haste makes waste
6. Respect people	There is one best way
7. Optimize the whole	Optimize by decomposition

The principles are explained using new anecdotes and examples to reinforce anything the reader might have missed from the first book. The myths are particularly intriguing to those who may recognize in them their own experiences. Here are some snippets of those myths:

Myth – Early specification reduces waste –

[1] Although Deming is not mentioned in this chapter, the book sweeps the trifecta of the items noted above by devoting an entire section in chapter 6 to Deming and his philosophy.

"The reason we develop all of this extra code lies in a game we play with our customers. We set up the rules:

> "Will you, valued customer, please give us a list of everything you want the software to do? We will write it down and ask you to sign off on the result. After that, if you want changes or additions, you will have to go through an arduous process called 'change management' to get a change approved. So you had better think of everything you want right now, because we have to know about all of it at the beginning in order to develop good software.

"Is it any wonder that our customers throw everything, including the kitchen sink, into their list of requirements? Far too often the game of scope control has the opposite effect – it creates scope bloat." (p. 24)

Myth – The job of testing is to find defects –

"The job of tests, and the people that develop and run tests, is to *prevent* defects, not to find them. A quality assurance organization should champion processes that build quality into the code from the start rather than test quality in later. This is not to say that verification is unnecessary. Final verification is a good idea. It's just that finding defects should be the exception, not the rule, during verification. If verification routinely triggers test-and-fix cycles, then the development process is defective.

"In Mary's plant, common wisdom held that 80 percent of the defects that seemed to be caused by people making mistakes were actually caused by a system that allowed the mistakes to happen. Therefore, the vast majority of defects were actually management problems." (p. 28)

Myth – Haste makes waste –

"In a quest for discipline, many organizations develop detailed process plans, standardize work productions, workflow documentation and specific job descriptions. This is often done by a staff group that trains workers in the process and monitors conformance. The goal is to achieve a standardized, repeatable process which, among other things, makes it easy to move people between projects. But a process designed to create interchangeable people will not produce the kind of people that are essential to make fast, flexible processes work." (p. 35)

Myth – There is one best way –

"The task of finding and institutionalizing the 'one best way' to do every job was handed over to industrial engineers in most American industries. Forerunners of the so-called process police, these industrial engineers often created standards without really understanding the work and then enforced them without entertaining the possibility that there may be a better way.

"Processes should be improved by the work team doing the job. They need the time, the charter, and the guidance to tackle their problems, one at a time, biggest problem first. This never-ending continuous improvement process should be found in every software development organization." (p. 38)

Myth – Optimize by decomposition –

"We can't measure everything, and because we have to limit our measurements, some things invariably fall between the creaks. For example, we tend to measure project performance based on cost, schedule, and scope targets. But these measures don't take quality and customer satisfaction into account. If a conflict arises, they lose out." (p. 40)

3. Value

This chapter explores those things in an organization that constitute value. It focuses on the value of satisfied customers and on the concept of leadership. It features Google (1999) as the example in explaining tracking the value stream from Concept to Cash, offering:

> "By all measures, Google is a very successful organization (as of this writing), and more importantly, its products have changed the way the world works. Google continually introduces new products despite the fact that it doesn't have a long-term product map and it expects technical staff to spend 20 percent of their time on their own projects." (p. 45)

4. Waste

This chapter features clothing company Zara, of Spain, as the example in how effectively waste can be reduced. As in the first book, the Seven Wastes are presented, but some changes were introduced as shown in the comparison below: (p. 74)

The Seven Wastes of Manufacturing	**The Seven Wastes of Software Development (book 1)**	**The Seven Wastes of Software Development (book 2)**
1. Inventory	1. Partially Done Work	1. Partially Done Work
2. Extra Processing	2. Extra Processes	2. Relearning
3. Overproduction	3. Extra Features	3. Extra Features
4. Transportation	4. Task Switching	4. Handoffs
5. Waiting	5. Waiting	5. Delays
6. Motion	6. Motion	6. Task Switching
7. Defects	7. Defects	7. Defects

It gives these as examples of partially done work: (p. 74)

- Uncoded documentation
- Unsynchronized code
- Untested code
- Undocumented code
- Undeployed code

On the topic of inventory, they provide this sidebar:

> **"We have a big inventory of documentation but we can't change that because government regulations require us to have a Software Requirements Specification (SRS) and traceability to code.**
>
> "If you must have an SRS and traceability to code, then consider writing as much of the SRS as executable tests as possible. Consider using FIT (Framework for Integrated Tests) or a similar acceptance-testing tool, to write specifications-by-example. A tool like this can give you traceability of tests to the code for free. If you run the tests every day and save the output in your configuration management system, you will have a record of exactly which tests passed and which ones didn't at any point in time. Regulators will love this." (p. 75)

On relearning …

> "Rediscovering something we once knew and have forgotten is perhaps the best definition of 'rework' in development. We know we should remember what we have learned. Nevertheless,

> our approach to capturing knowledge is quite often far too verbose and far less rigorous that it ought to be." (p. 76)

On handoffs …

> "Handoffs are similar to giving a bicycle to someone who doesn't know how to ride. You can give them a big instruction book on how to ride the bike, but it won't be of much help. Far better that you stay and help them experience the feeling of balance that comes with gaining momentum. Then give some pointers as they practice starting, stopping, turning, going down a hill, going up a hill. Before long your colleague *knows* how to ride the bike, although she can't describe how she does it. This kind of knowledge is called tacit knowledge, and it is very difficult to hand off to other people through documentation." (p. 77)

On defects …

> " … the real reason for moving testing to the beginning of development is deeper than mistake-proofing. Acceptance tests are best when they constitute the *design* of the product and match that design to the structure of the domain. Unit tests are best considered the *design* of the code; writing unit test before writing code leads to simpler, more understandable, and more testable code. These tests tell us exactly and in detail how we expect the code and the ultimate product to work. As such, they also constitute the best documentation of the system, documentation that is always current because the tests must always pass." (p. 82)

Also covered here are the benefits of mapping a value stream and provides 4 examples of typical value stream maps. The authors profess:

> "The objective of lean is to reduce the development timeline by removing non-value-adding wastes. Value stream maps have proven remarkably effective at exposing waste, because delays in the flow are almost always a sign of significant waste." (p. 83)

5. Speed

This chapter features healthcare industry software company PatientKeeper as the example of how rapidly a product can be developed and released. It covers the concepts of queuing theory and reducing cycle time. In the section on Variation and Utilization they state:

> "If you have ever wondered why Google chose to dedicate 20 percent of its scientists' and engineers' time for work on their own projects, … cycle time starts to increase at just above 80 percent utilization, and this effect is amplified by large batches (high variation). Imagine a group of scientists who study queuing theory for a living. Suppose they find themselves running a company that must place the highest priority on bringing new products to market. For them, creating 20 percent slack in the development process would be the most logical decision in the world. It's curious that observers applaud Google for redundant servers but do not understand the concept of slack in a development organization." (pp. 101-102)

6. People

This chapter features the Boeing 777 project (1988) as the example in how effectively to inspire and tap the potential of the people who work on a project. It covers teams and what makes them, expertise, leadership, self-directing work, and worker incentives and compensation.

The section A System of Management provides this poignant passage on the importance of people:

> "Lean is a management system that creates engaged, thinking people at every level of the organization and most particularly at the front line. If you implement all of the lean principles save one – respect for people – you will reap only a shadow of the potential advantages that lean can bring. If you implement only one principle – respect for people – you will position the people in your organization to discover and implement the remaining lean principles." (p. 117)

The section on W. Edwards Deming goes into some extended detail on his famous 14 Points and his System of Profound Knowledge, including these snippets:

> "Alan Mulally created a management system for the [Boeing] 777 straight out of Deming's playbook at a time when many companies throughout the US were discovering the power of Deming's ideas." (p. 123)

> "The innovation in management thinking that Deming brought to the world is this: When things go wrong, the cause is almost always inherent in the system, and therefore it is a management problem. Managers who push harder on people to meet deadlines or reduce defects are ignoring the system problems and will usually make things worse. Instead, leaders must envision and institute fundamental changes that address the systemic causes of current problems." (pp. 123-124)

The section on Incentives issues this warning:

> "If you have a ranking system in your company, a lean initiative will be hollow at best. The behaviors that ranking systems encourage are competition, hiding information so as to look good, and hiding problems so as not to look bad." (p. 143)

7. Knowledge

This chapter features Rally Software Development as the example in how to create knowledge. It covers how to keep track of what you know and learn, set-based design, refactoring and a disciplined approach to problem solving.

The section titled What, Exactly, Is Your Problem? provides this counsel :

> "Lean initiatives must always start with a clear vision of how you make money and a sharp understanding of the most critical problem that is keeping you from doing so." (p. 153)

The section titled A Scientific Way of Thinking compares the Toyota Production System problem-solving method with the Deming PDCA cycle[2] (p. 155):

Toyota	Deming
1. Frame the problem.	1. PLAN
2. Look for the root cause.	
3. Propose a countermeasure.	
4. Specify the expected result.	
5. Implement the counter measure.	2. DO
6. Verify the results.	3. CHECK
7. Follow up/Standardize.	4. ACT

Keeping Track of What You Know offers this paragraph supporting a point I have been championing to rebuff policies requiring technical documentation to contain information I consider to be nothing more than extraneous fluff:

> "In software development, the tests and code are often just the right combination of rigor and conciseness to document the knowledge embedded in the software. But experiments tried and options investigated on the way to making decisions about the product under development are easily forgotten, because they never make it into the software. Just writing something down does not necessarily turn it into knowledge, and information is just as easily lost in a sea of excess documentation as it is lost through lack of documentation." (p. 156)

and this:

> "Early in their careers, Toyota engineers learn the discipline of condensing complex thinking to a single A3 [11" x 17"] sheet of paper. This forces people to filter and refine their thoughts so that anyone reading the A3 report will have all of their questions answered by reading a single sheet of paper. Different A3 reports have different purposes, but all of them capture critical knowledge in a way that is easy to store in a database, easy to post in a work area, easy to send to a manager and is easy to incorporate into future experiments" (p. 157)

In Set-Based Design is this justification:

> "What we think of as waste is almost entirely dependent on how we frame the problem we are trying to solve. Remember the emergency response Teams A, B and C [referring to an example of having three different teams concurrently addressing a problem where A represented the shorter-time-to-complete minimal solution and C the longer-time-to-complete ultimate solution]. We have been asked 'Why not put people from Team A onto Team B and increase the chances of getting a better solution earlier?' Recall, however, that this was an emergency and *failure was not an option*. If we had said *this was a matter of life and death and failure would mean someone dies*, would Team A still seem like a waste?" (p. 164)

In the section Refactoring is a snippet stating:

> "The rationale behind refactoring is this:

[2] The PDCA cycle also is known as the PDSA cycle, where Study replaces Check. Furthermore, as partially raised in a footnote on page 121, though commonly known as the Deming cycle, it originally was conceived by Walter A. Shewhart, Deming's mentor, and Deming himself had referred to this as the Shewhart Cycle.

1. Adding features before we are sure they are needed increases the complexity, making it the worst form of waste in software development. Therefore, most features should be added incrementally.
2. When adding features and changes to an existing code base it is essential not to add complexity.
3. Refactoring reduces the complexity of a code base simplifying its design. This allows new features to be accommodated with minimum complexity." (pp. 164-165)

This sage advice appears in the section Problem Solving:

> "It does no good to have a software development process if it is not accompanied by a way for those who struggle with its idiosyncrasies to fix them. Every team should have a regular time to systematically find and fix the things that make their lives difficult." (p. 168)

8. Quality

This chapter features the Polaris Submarine Program (1957 - 1965) of the US Department of Defense as the example in producing quality. It covers iterations, discipline, standards, mistake-proofing and test-driven development.

In the section on Release Planning is this assessment of it:

> "A release plan gives you a basis for testing technical and market assumptions before investing a large amount of money in detailed plans. You can get an early reading of the plan's feasibility by starting implementation and tracking progress. By the first release you will have real data to forecast the effort needed to implement the entire project far more accurately than if you had spent the same six weeks generating detailed requirements and plans." (p. 181)

In the section on Architecture is this perspective on it:

> "In successful products whose lifetimes span many years, a major architectural improvement can be expected every three years or so, as new applications are discovered and capabilities are required that were never envisioned in the original architecture. It is time to abandon the myth that architecture is something that must be complete before any development takes place." (p. 182)

The section on Iterations contains good information for establishing an iterative development process but includes two diagrams, one an example of iterative development, the other of nested cycles, which I found to be more confusing than helpful.

The section on Standards contains these two gems, addressing two aspects of software development I have come to regard as mostly ineffective in the way they are implemented at organizations where I have worked:

> "… standards are useless on paper; they have to be used consistently to be valuable. In a lean environment, standards are viewed as the current best way to do a job, so they are always followed. The assumption, however, is that there is *always* a better way to do things, so everyone is actively encouraged to challenge every standard. Any time a better way can be found and proven to be more effective, it becomes the new standard. The discipline of following – and constantly changing – standards should be part of the fabric of an organization." (p. 194)

> "'Are code reviews waste?' people often ask in our classes. Good question. We think that using code reviews to enforce standards or even to find defects is a waste. Code analyzers and IDE checks are the proper tools to enforce most standards, while automated testing practices are the proper way to avoid most defects. Code reviews should be focused on a different class of

problems. For example, the code of inexperienced developers might be reviewed for simplicity, change tolerance, absence of repetition, and other good object-oriented practices. … Some policies that require code reviews prior to check-in create a large accumulation of partially done work waiting for review, but in a lean environment this is an unacceptable situation that should be avoided aggressively." (pp. 194-195)

This snippet is taken from the section on Test-Driven Development:

> "… [Toyota's] Shigeo Shingo taught that there are two kinds of tests: tests that find defects after they occur and tests to prevent defects. He regarded the first kind of tests as pure waste. The goal of lean software development is to prevent defects from getting into the code base in the first place, and the tool to do this is test-driven development.
>
> "'We can't let testers write tests before developers write code', one testing manager told us. 'If we did that, the developers would simply write the code to pass the tests!'. Indeed, that's the point." (p. 198)

This question appears at the end of the chapter:

> "6. Who is responsible for setting standards? Who should be? How closely are standards followed? How easily are they changed? Is there a connection between the last two answers in your organization?" (p. 205)

9. Partners

This chapter features the Open Source Software Community as the example in establishing and effectively using partnerships. It covers synergy, outsourcing, joint ventures and contracts, and contained this comment on what constitutes a team:

> "Going back to Deming's 14 points, point 12 is probably the most important: Eliminate barriers that rob people of their right to pride in workmanship. Artificial deadlines, managers with no idea about what good code really is, sloppy code bases and coding practices, no way to see if new code works, no idea what the users really want, individual rewards for team contributions – all of these rob every member of the development team of pride in workmanship. Pride builds commitment, and without mutual commitment, a group really isn't a team." (p. 210)

10. Journey

This chapter features Toyota as the example in developing " … a long-range strategy for creating and sustaining competitive advantage." (p. 224) It covers long-term perspectives, the quality initiatives Six Sigma and Theory of Constraints, and presents a roadmap for the future.

The section A Long-Term Perspective includes this observation:

> "Organizations that have become successful tend to establish their habits in the days when they are growing rapidly and there is plenty of market demand. But eventually all growth engines run out of fuel, and past success no longer points the way to further growth. At this point, the organizations that have developed the capability to learn and adapt are the ones that survive. De Geus [Arie, author of The living Company: Habits for Survival in a Turbulent Business Environment] believes it is the ability of managers to envision the future that enables them to adapt to that future before it is too late; but unfortunately, mangers often develop 'future-blindness,' preferring to stay on the course that has led to success in the past." (p. 226)

This suggested introspection is from the section Centered on People:

> "The question to resolve before you embark on a lean initiative is this: What do you *really* believe about people? Consider your attitude toward process. Do you think that a well-documented process that everyone follows without question is the path to excellence? Or do you think that the reason to standardize a process is so that people doing the work have something solid to question and change? Lean principles are solidly based on the second viewpoint". (p. 228)

The section Hypothesis contributes this as a starting point:

> "The most effective way to get started with lean development is:
> - Train team leaders and supervisors (in preference to change agents).
> - Emphasize on-the-job thinking (rather than documentation).
> - Measure a few key indicators (as opposed to many data points)." (p. 234)

The roadmap for getting started with lean offers these 10 points (p. 242):

1. Begin where you are
2. Find your biggest constraint
3. Envision your biggest threat
4. Evaluate your culture
5. Train
6. Solve the biggest problem first
7. Remove accommodations
8. Measure
9. Implement
10. Repeat the cycle

Summary

This is a brilliant successor to their first book. Like the first book, it is relatively short and easy to read. The bibliography lists 8 full pages of other publications referenced throughout the text, ranging from Frank B. Jr. and Earnestine Gilbreth's Cheaper by the Dozen (1948) to Nancy Van Schooenderwoert's Embedded Agile Project by the Numbers with Newbies (2006), with more than half of these having been published since 2000.

Notable passages

> "If people in your organization instinctively work around problems, they have the *wrong reflexes!* Brainstorm what it will take to develop a culture that does not tolerate abnormalities, whether it is a broken build or a miscommunication, a failed installation or code that is not robust enough to hold up in production." (p. 18)

> "I did not run into the term 'waterfall' until 1999 when I started working on a government project. … I was puzzled by the waterfall approach because I couldn't understand how it could possibly work; and as a matter of fact, it didn't work. As I compared my experience working on complex, successful projects to the prescribed waterfall approach which failed on a relatively small project, I decided to write a book about what really works. In that book [Lean Software Development: An Agile Toolkit] we outline seven principles of software development, which are summarized [in this book]. – Mary Poppendieck" (p. 22)

> "One of the puzzling aspects of 'waterfall' development is the idea that knowledge, in the form of 'requirements', exists prior to and separate from coding. Software development is a knowledge-creating process. While an overall architectural concept will be sketched out prior to coding, the validation of that architecture comes as the code is being written. In practice, the detailed design

of software always occurs during coding, even if a detailed design document was written ahead of time. An early design cannot fully anticipate the complexity encountered during implementation, nor can it take into account the ongoing feedback that comes from actually building the software. Worse, early detailed designs are not amenable to feedback from stakeholders and customers. A development process focused on creating knowledge will expect the design to evolve during coding and will not waste time locking it down prematurely." (pp. 29-30)

"It is important to have a development process that encourages systematic learning throughout the development cycle, but we also need to systematically improve that development process. Sometimes in the search for 'standard' processes we have locked our processes up in documentation that makes it difficult for development teams to continually improve their own processes." (p. 31)

"Just about every start-up company with a single software product is nimble and can quickly respond to customers and adapt to change. But after a couple of years of success, software companies often start to slow down and become unresponsive. They have developed a complex code base, a complex array of products, and a complex organizational structure. Unless they get the complexity under control quickly at this point, it will gradually strangle the very responsiveness that gave the company its initial competitive advantage." (p. 69)

"In manufacturing, people have learned that a lot of in-process-inventory just gums up the works and slows things down. Somehow we don't seem to have learned this same lesson in development. We have long release cycles and let stuff accumulate before releasing it to production. We have approval processes that dump work into an organization far beyond its capacity to respond. We have sequential processes that build up an amazing amount of unsynchronized work. We have long defect lists. Sometimes we are even proud of how many defects we've found. This partially done work is just like the inventory in manufacturing – it slows down the flow, it hides quality problems, it grows obsolete, usually pretty rapidly." (p. 105)

"Rewriting legacy software is an attractive approach for managers who dream of walking into their office one day and finding that all of their problems have disappeared as if by magic, and the computer now does all of the things that they could never do with the former system. The problem is, this approach has a dismal track record. Attempting to copy obscure logic and convert a massive database – which is probably corrupted – is very risky. Furthermore, as we have seen, perhaps two thirds of the features and functions in the legacy system are not used and not needed. Yet too many legacy conversions attempt to import the very complexity that caused the legacy system to resist change in the first place." (p. 166)

"One of [the] most rewarding ways to mistake-proof development is to automate routine tasks. Even small teams should automate everything they can, and even occasional tasks should be candidates for automation. Automation not only avoids the eventual mistakes that people will always make, it shows respect for their intelligence. Repetitive tasks are not only error prone; they send the message that it's OK to treat people like robots. People should not be doing things by rote; they should [be] thinking about better ways of doing their job and solving problems." (pp. 197-198)

"The development environment of 30 years ago, 20 years ago, even 10 years ago is entirely different than it is today, and the accommodations we made to deal with limited memory, processor power, communication capability, and software options are no longer appropriate. … In an environment that is changing so rapidly, we would do well to be alert for the accommodations that we continue to make for constraints that are no longer present." (p. 233)

"Great solutions delight customers. True, basic needs of customers must be met, and performance must be in line with competitors. But the goal of lean development should be to find ways to delight customers by understanding their situation deeply and solving their problems completely. In the book *The ultimate Question*, Fred Reichheld claims that companies that

delight customers gain a sustainable competitive advantage, while companies that annoy customers will lose these customers the moment a more attractive alternative is available." (p. 241)

Useful links

- "Agile Testing Directions" by Brian Marick – www.testing.com/cgi-bin/blog/2003/08/21-agile-testing-project-1 (p. 199); available in August 2009 at http://www.exampler.com/testing-com/agile/
- Framework for Integrated Tests (FIT) and Fitness – www.fitnesse.org; (pp. 75, 150)
- Jidoka (aka Autonomation) video – www.toyota.co.jp/en/vision/production_system/video.html (p. 17)
- "Lean or Six Sigma" by Freddie Balle and Michael Balle – www.lean.org/library/leanorsigma.pdf (p. 5)
- "The Polaris: A Revolutionary Missile System and Concept" by Norman Polmar – www.history.navy.mil/colloquia/cch9d.html (p. 178)
- Poppendieck.LLC, web site for Mary and Tom Poppendieck – www.poppendieck.com - Mary and Tom collaborated on yet another book on the subject of Lean Software Development, this one titled "Leading Lean Software Development: Results are Not the Point", to be available in October 2009.
- Process capacity – www.lean.org (p. 98)
- "The Roots of Lean" by Jim Huntzinger – www.lean.org/Community/Resources/ThinkersCorner.cfm (p. 235); available in August 2009 at http://www.lean.org/common/display/?o=106
- "TPS [Toyota Production System] vs. Lean Additional Perspectives" by Art Smalley – www.superfactory.com/articles/Smalley_TPS_vs_Lean_Additional_Perspectives.htm. (p. 152); available in August 2009 at http://artoflean.com/documents/pdfs/TPS_versus_Lean_additional_perspectives_v3.pdf
- "TPS [Toyota Production System] vs. Lean and the Law of Unintended Consequences" by Art Smalley – www.superfactory.com/articles/smalley_tps_vs_lean.htm. (pp. 153; 229; 236); available in August 2009 at http://www.superfactory.com/articles/featured/2005/0512-smalley-tps-lean-manufacturing.html