Lean Software Development – An Agile Toolkit

Mary Poppendieck and Tom Poppendieck; Addison-Wesley, 2003, 203 pages
ISBN 0-321-15078-3

Book report
Jim McDonough, July 4, 2009

Review

This book takes the "lean manufacturing" principles of the Toyota Production System and adapts them to the software development industry. The Toyota Production System, the vision of Taiichi Ohno, emerged in response to the challenge of how to produce cars in small quantities and keep them as inexpensive as those made through mass production processes. The fundamental principle of lean manufacturing is to eliminate waste. As described in the Introduction, the book contains seven chapters corresponding to the seven lean principles and presents "thinking tools" for translating each lean principle into agile practices.

The book describes these seven lean principles:

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.

It proclaims that eliminating waste is the most fundamental lean principle, the one from which all other principles follow (p. 2), and compares the seven wastes found in manufacturing with seven comparable wastes of software development (p. 4):

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

It presents 22 thinking tools – an Agile Toolkit – to be used to administer the seven lean principles (p. 117):

1. Seeing Waste
2. Value Stream Mapping
3. Feedback
4. Iterations
5. Synchronization
6. Set-Based Development
7. Options Thinking
8. Last Responsible Moment
9. Making Decisions
10. Pull Systems
11. Queuing Theory
12. Cost of Delay
13. Self Determination
14. Motivation
15. Leadership
16. Expertise
17. Perceived Integrity
18. Conceptual Integrity
19. Testing
20. Refactoring
21. Measurements
22. Contracts

This book provides a refreshing perspective for anyone who has worked in a development environment encumbered by bureaucracy. Throughout the book are sidebars giving real examples of development projects thwarted by dogmatic adherence to rigid processes of the past, and of success stories for having embraced lean thinking and jettisoning those tasks contributing no value. A recurring theme is the comparison between software development cultures that are "change intolerant" to those that are "change tolerant". Often it is cited that customers:

- Usually do not know what they want
- Can be hesitant to sign-off on requirements and specifications because they are afraid they will have no opportunities to make changes
- Historically have been several levels removed from developers implementing the features
- Generally get a good sense of progress and usability when they see working code

Through both explanation and example this book endorses the use of techniques not yet commonplace in most development organizations:

- Iterative development
- Time-boxing
- Establishing a relationship of trust with customers and vendors
- Management as facilitator to developers rather than overlord
- Reduction of paperwork
- Elimination of development activities which provide no value to the customer
- Designing systems and developing code without fixed requirements and specifications
- Planning for changes in software design during its development
- Toleration of, even encouragement for, multiple design approaches and corresponding prototypes as development progresses

On page 179 is a set of cautions qualifying the seven lean principles:

- **Eliminate waste** does not mean throw away all documentation.
- **Amplify learning** does not mean keep on changing your mind.
- **Decide as late as possible** does not mean procrastinate.
- **Deliver as fast as possible** does not mean rush and do sloppy work.
- **Empower the team** does not mean abandon leadership.
- **Build integrity in** does not mean big, upfront design.
- **See the whole** does not mean ignore the details.

Eliminate Waste

This chapter explains how to see waste and reduce it. Few things contribute more to improving quality than this simple activity. It proposes using a technique called "value stream mapping" to determine where there is value and where there is waste in a process.

Amplify Learning

My expectation for this chapter was to read about how those in a development organization share their knowledge with each other – essentially raising the skill level of the staff. While such knowledge exchange is a part of amplifying learning, the focus here is on learning from doing – as in running a test of a prototype to determine what can be learned from the result. Essentially this aspect of learning corresponds to the "S" in the Shewhart PDSA cycle (Plan-Do-Study-Act). The more often that such cycles

can be run, the more amplification of the learning to be gained from the process. Not surprisingly, this chapter introduced the tool “Iterations”.

Decide as Late as Possible

This chapter gives credibility to the process of “Set-based development”, where multiple approaches to a design are concurrently developed. At some point the constraints inherent in the desired features will begin to eliminate some of these approaches as more is learned about the capabilities of each approach. As such, a later decision may be made about which is the best approach. Compare this with the waterfall method of software development, where only one approach is pursued and this design is decided well before any knowledge can be gained about its suitability.

Deliver as Fast as Possible

This chapter is not about how to make developers crank out code faster but how to establish a system of code delivery which makes the most effective use of developers time. It introduces the tools “Pull Systems”, where developers are not so much assigned tasks (pushed onto them) as they are expected to pull a task from a list of tasks awaiting completion; “Queuing Theory”, which discusses how to eliminate bottlenecks; and “Cost of Delay”, which presents alternatives to determining the true costs associated with software development.

Empower the Team

This chapter introduces the tool “Self-Determination” by giving the example of the General Motors plant closure in Fremont, California in 1982, and resurrected two years later by New United Motor Manufacturing, Inc. (NUMMI), a joint venture between Toyota and GM. Until 1982 this plant was a dismal failure, but after NUMMI began operations with the same employees who had worked there prior to the 1982 closing it was hailed as an unqualified success after only two years. The new Toyota management team empowered the employees to make their own decisions about how the work was to be performed.

Other tools “Motivation, “Leadership” and “Expertise” all reinforce the idea that most workers know fairly well how to do their jobs and do them well, and neither need nor appreciate the micromanagement foisted upon them by those who believe workers need to be told how to do their jobs.

On page 109 appears this passage:

> “Also important is a mechanism for sharing ideas and improving designs, perhaps by using pair programming, code reviews, or similar approaches.”

After reading twice cover-to-cover, this is the only reference to “code review” I found in the entire book. It puts into perspective the relative importance of code review as perceived by expert practitioners of lean software development.

Build Integrity In

This chapter distinguishes between the tools “Perceived Integrity” – what the customer thinks about the usefulness of the system – and “Conceptual Integrity” – how well the software pieces fit together and interact compatibly. It introduces the tool “Refactoring”, which is the act of rewriting software to maintain Conceptual Integrity. I did not know this had a name, but I have been doing this for years, usually under the radar at most installations where I have worked, primarily due to the perception by management that such activity is a waste of time.

Indeed, a sidebar titled “A Reward for Developers” states the following:

> “Microsoft keeps the same team working on a product such as Excel over multiple releases. After the team has worked hard to complete a release, the members are rewarded with a couple of months in which they are allowed to clean up the underlying structures of the code that bothered them the most while working on the release.” (p. 142)

Even at Microsoft the prevailing management attitude seems to be that refactoring constitutes not a strategy for assuring the maintainability of the code, but a *reward* bestowed upon developers who then are *allowed* to make such improvements.

Five characteristics of Conceptual Integrity are offered, which, once eroded, signal the time to refactor:

1. Simplicity
2. Clarity
3. Suitability for Use
4. No Repetition
5. No Extra Features

The claim is made that refactoring is not rework and provides this explanation of its importance:

> “Concurrent development means that the design of the product emerges throughout the development process. Improving a design during the development process is most certainly not rework; it is good design practice. Okay, you say, but there isn’t time to stop development to improve the design. We would argue that there isn’t time *not* to refactor. Work will only go slower as the code becomes complex and obscure.” (p. 144)

This chapter also introduces the tool “Testing” and takes this to new levels of importance, postulating that a robust test bed is far more important to a development organization as proof of software integrity than many other deliverables, going so far as proposing that a comprehensive automated testing system is more valuable to subsequent maintenance efforts than any documentation since it can more accurately reveal the as-built system as well as serve as a starting point for new development.

See the Whole

This chapter eloquently expresses the concept of “seeing the whole” by discussing Lance Armstrong’s four consecutive victories in the Tour De France between 1999 and 2002.[1] The Tour De France is a grueling 21-stage bicycle race, but winning the race is not about winning individual stages. In 2002 Lance won only one of the 21 stages, and never more than 4 stages during that 4-year period. Had he concentrated on winning each stage instead of “seeing the whole” race as a set of 21 stages history might have been different.

The tools “Measurements” and “Contracts” are introduced here. After stating three basic patterns to systems thinking – limits to growth; sifting the burden; suboptimization – it focuses on suboptimization, which is the optimization of a local area or staff at the expense of the larger area or company. Superstition and habit are two reasons suggested for why suboptimization occurs, and the radical concept of “The Unimportance of Cost and Schedule” is explained in a sidebar, concluding with:

> “Who needs cost and schedule control when you are navigating from a business plan? Many people are jarred by the idea that a company that develops new products so successfully does not manage product development projects by cost or schedule. Why? They have fallen into the habit of thinking that cost and schedule are the important things in managing a project. It’s hard

[1] Lance since went on to win the next three years, took three years off, and as of this writing the 2009 Tour De France is underway with Lance Armstrong competing again.

for them to think of these as suboptimizing measurements. Yet a focus on cost and schedule would have distracted us from our ultimate objective: Develop and commercialize a profitable new product that meets a customer need and has a competitive advantage." (pp. 158-159)

This topic of suboptimization reminds me of an illustrative anecdote recounted by W. Edwards Deming in his book The New Economics:[2]

> **"An example of destruction of a system.** Engine and transmission both had electrical components in them. An engineer with great knowledge redesigned some of the components, and found that by putting other and different electrical components in the engine, none would be needed in the transmission. The following table portrays the alternatives.
>
> Electrical components
>
> | Status | Engine | Transmission | Both |
> |---|---|---|---|
> | As is | $100 | $80 | $180 |
> | Proposed | $130 | $ 0 | $130 |
> | Gain from proposal | | | $ 50 |
>
> The proposal was rejected by the financial people associated with the engine, because the proposal would increase by $30 the cost of the engine. Their job was to decrease costs of the engine, not to increase them. That the proposal would decrease the overall cost of the whole company by $50 was not a factor to consider by the financial people associated with the engine. Their job was with the engine, not the vehicle. The engine to them was an individual profit center."

The section here on "Contracts" is interesting since it clarifies much of the behavior I have witnessed in software development and maintenance organizations toward customers.

Summary

This book presents a good start to those unfamiliar with lean development concepts. It is relatively short and easy to read. The bibliography lists 7 full pages of other books referenced throughout the text, from as far back as Winslow Taylor's Principles of Scientific Management (1911) to as recent as Eric Evans' Domain Driven Design (2003). At the time of publication, the two authors had between them nearly 40 years of experience in information technology projects, with Mary Poppendeick serving as the Managing Director of the Agile Alliance.

Notable passages

> "Others and I have made attempts to provide a theoretical underpinning to agile processes. I've referred back to my research in industrial process control theory, which friends of mine at DuPont's Advanced Research Facility helped me understand." – Ken Schwaber (p. xv)
>
> "I heard, for example, that detailed process definitions were needed so that 'anyone can program', while lean manufacturing focused on building skill in frontline people and having them define their own processes." (p. xvii)
>
> "We observe that some methods still considered standard practice for developing software have long been abandoned by other disciplines. Meanwhile, approaches considered standard in product development, such as concurrent engineering, are not yet generally considered for software development" (p. xxii)

[2] Deming, "The New Economics for Industry, Government, Education", 2nd edition, 1994, p. 67.

“In 1970 Winston Royce wrote that the fundamental steps of all software development are analysis and coding. … With our definition of waste, we can interpret Royce’s comment to indicate that *every step in the waterfall process except analysis and coding is waste*.” (p. 4)

“Do you ever ask, Is all that paperwork really necessary? Paperwork consumes resources. Paperwork slows down the response time. Paperwork hides quality problems. Paperwork gets lost. Paperwork degrades and becomes obsolete. Paperwork that no one cares to read adds no value.” (p. 5)

“Traditional project management approaches often consider feedback loops to be threatening because there is concern that the learning involved in feedback might modify the predetermined plan. The conventional wisdom in project management values managing scope, cost, and schedule to the original plan. Sometimes this is done at the expense of receiving and acting on feedback that might change the plan; sometimes it is done at the expense of achieving the overall business goal. This mental mode is so entrenched in project management thinking that its underlying assumptions are rarely questioned. This might explain why the waterfall model of software development is so difficult to abandon.” (p. 25)

“The development team must be free to accept only the amount of work for an iteration that team members believe they can complete within the time-box. Customers will probably want to load iterations with lots of features, but it is important to resist the temptation to be accommodating at the expense of setting unreasonable expectations. If iterations are short and delivery is reliable, customers should be content to wait for the next iteration. If a development team over commits – which often happens to inexperienced teams – it is best to deliver some of the features on time rather than all of them late.” (p. 29)

“Since customers often don’t know exactly what they want at the beginning of a project, they tend to ask for everything they think they might need, especially if they think they will get only one shot at it. This is one of the best ways we know to increase the scope of a project well beyond what is necessary to accomplish the project’s overall mission.” (p. 32)

“Sometimes an entire suite of tests is run, and sometimes, especially when tests are manual, only some tests are run. The general principle is that if builds and test suites take too long, they will not be used, so invest in making them fast.” (p. 35)

“Programming is a lot like die cutting. The stakes are often high, and mistakes can be costly, so sequential development, that is, establishing requirements before development begins, is commonly thought of as a way to protect against serious errors. The problem with sequential development is that it forces designers to take a depth-first rather than a breadth-first approach to design.” (p. 48)

“No one seemed to recognize that the sequential process could actually be the *cause* of the high escalation ratio.” (p. 50)

“Lean software development delays freezing all design decisions as long as possible, because it is easier to change a decision that hasn’t been made.” (p. 52)

“… if systems are built with a focus on getting everything right at the beginning in order to reduce the cost of later changes, their design is likely to be brittle and not accept changes readily.” (p. 52)

“The prevailing paradigm has been a predictive process: Software development should be specified in detail prior to implementation because if you don’t get the requirements nailed down and the design right, it will surely cost a lot to make changes later. This paradigm may work in a highly predictable world. However, if there is uncertainty about what customers really need,

whether their situation will change, or where technology is moving, then an adaptive approach is a better bet." (p. 56)

"The notion that a design must be complete before it is released is the biggest enemy of concurrent development." (p. 57)

"Sometimes it seems that there are not enough experienced people available to allow intuitive decision making, and therefore rational decision making is the better approach. We strongly disagree. It is much more important to develop people with the expertise to make wise decisions than it is to develop decision-making processes that purportedly think for people." (p. 62)

"Thus, simple rules are a key mechanism to enable people to *decide as late as possible*." (p. 65)

"There are two ways to assure that workers make the most effective use of their time. You can either tell them what to do or set things up so they can figure it out for themselves. In a fast-moving environment, only the second option works. People who deal with fluid situations, such as emergency workers and military personnel, do not depend on a remote commander to tell them how to respond to the latest development. They figure out how to respond to events with the other people who are on the scene. When things are happening quickly, there is not enough time for information to travel up the chain of command and then come back down as directives. Therefore, methods for local signaling and commitment must be developed to coordinate work. One of the key ways to do this is to let customers' needs *pull* the work rather than have a schedule *push* the work." (p. 71)

"All too often, a software development team is told that it must meet cost, feature, and introduction date objectives simultaneously; there can be no tradeoffs. This sends two messages to the development team: 1) Support costs aren't important because they weren't mentioned; 2) When something has to give, make your own tradeoffs." (p. 84)

"One of the fastest ways to kill motivation is what is called in the U. S. Army a *zero defects mentality*. A zero defects mentality is an atmosphere that tolerates absolutely no mistakes; perfection is required down to the smallest detail. The army considers a zero defects mentality to be a serious leadership problem, because it kills the initiative necessary for success on a battlefield." (p. 108)

"Even a highly motivated team will only work so long before members need to feel they have accomplished something. This reaffirms the purpose and keeps everyone fired up. If there is no other reason to develop software in iterations – and there are many! – this is a compelling reason by itself. " (p. 109)

"Software development leaders will not flourish in an organization that values process, documentation, and conformance to plan above all else. An organization will get what it values, and the Agile Manifesto does us a great service in shifting our perception of value from process to people, from documentation to code, from contracts to collaboration, from plans to action." (p. 115)

"Developers appreciate reasonable standards, especially if they have a hand in developing them and keeping them current." (p. 122)

"The problems with sequential development … It's the programmers who are going to be making day-to-day decision on exactly how to write the code. They are two or three documents away from an understanding of the customer perception of system integrity. At each document hand-off a considerable amount of information has been lost or misinterpreted, not to mention key details and future perspectives that were not obtained in the first place." (p. 130)

“One of the key approaches to incorporating change into an information infrastructure is to ensure that the development process itself incorporates ongoing change. One of the fears of those considering an iterative development approach is that later iterations will introduce capabilities that require change to the design. However, if a system is built under the paradigm that everything must be known up front so the optimal design can be found, then it will probably not be adaptable to change in the future. A change-tolerant design process is more likely to result in a change-tolerant system.” (p. 134)

“Maintaining institutional memory about a system is key to assuring its long-term integrity. There have been many attempts to use documentation created during design to do this. However, design documentation rarely reflects the system as it was actually built, so it is widely ignored by maintenance programmers. If this is the only purpose documentation serves, it was a waste to create it.” (p. 134)

“Just like in advertising, refactoring doesn’t cost, it pays.” (p. 144)

“A better name for [unit, system and integration] tests might be *developer tests*, because their purpose is to assure that code does what the developer intended it to do. … A better name for [acceptance] tests … is *customer tests*, since their purpose is to assure that the system will do what the customers expect it to do” (p. 145)

“It may seem like writing tests slows down development; in fact, testing does not cost, it pays, both during development and over the system’s lifecycle.” (p. 147)

“You will get more payoffs from an effective test program than from most other investments you might make. … What you need is the test suite for each application, developed as scaffolding for change during development. These tests, assuming you keep them healthy, constitute an accurate set of as-built documentation for all the applications in your environment. If each application has an up-to-date test suite to prove its integrity, you can test the entire environment before a change is released.” (p. 149)

“In fact, policies from the past may actually become today’s constraints.” (p. 154)

“… it is important to aggregate performance measurements rather than attribute them to individuals. … The vast majority of defects have their root cause in the development systems and procedures, and trying to attribute defects to individual developers is a case of shifting the burden.” (p. 160)

“… Toyota understands that a strong supplier network is far more beneficial to its interests than short-term gains that come from taking advantage of a supplier.” (p. 162)

“If you are faced with CMMI, we suggest you learn about the struggles of the U. S. military acquisition organizations to become more agile. Over the past decade, a series of directives and regulation have attempted to bring the same lean thinking to DoD acquisition, which makes U. S. military logistics among the best in the world. In late 2002 Deputy Secretary of Defense Paul Wolfowitz canceled earlier attempts and tried again to ‘ … create an acquisition policy environment that fosters efficiency, flexibility, creativity and innovation.’ He lists 30 principles and policies behind the new defense acquisition system. At the top of the list are many of the principles and tools found in this book: decentralized responsibility, processes tailored to each program, learning and innovation, reduced cycle time, collaboration, a total systems approach.” (p. 183)