Chapter 1. An Agile Developer's Guide to Lean Software Development

"Time is the most valuable thing a man can spend." —Theophrastus (372 BC–287 BC)

IN THIS CHAPTER

This chapter describes the basic principles for Lean Software Development, the notion of fast-flexible-flow in the development pipeline, the benefit of value stream mapping, and the way Lean guides Agile teams.

Takeaways

Key insights to take away from this chapter include

• Most errors come from our systems and not from people.

• Lean extends Agile to create a system that helps minimize work-in-process and maximize the speed at which business value is created.

• Lean principles suggest focusing on business value, speed, and quality.

LEAN

Lean is the name given to Toyota's method of both producing and developing cars. As software developers, we are doing neither, so why is it of interest to us? The reason is simple: The principles that underlie Toyota's methods are principles that in reality work everywhere. These are not panaceas, though. The principles are universal, although the specific practices may vary.

Principles are underlying truths that don't change over time or space, while practices are the application of principles to a particular situation. Practices can and should differ as you move from one environment to the next, and they also change as a situation evolves. (Poppendieck and Poppendieck 2006)

The principles that drive Lean can be applied to software. Doing so provides guidance for those who want to develop software more effectively. In this chapter, instead of describing Lean on its own, we describe Lean in terms of the Agile practices it suggests. This illustrates how Agile practices are manifestations of Lean principles. There is power to this understanding—when an Agile practitioner finds him- or herself in a situation where a standard Agile practice cannot be followed, the Lean principles can guide him or her to a better way. Lean principles also often highlight different issues than standard Agile practices do. By making certain things explicit, Agile practitioners will have more power at their disposal to improve their methods.

LEAN APPLIES TO MANY LEVELS OF THE ORGANIZATION

This book is written for anyone who leads or plays a role in an organization to define, create, and deliver technology solutions. When we refer to the "enterprise," we mean

All parts of the organization that are involved in the value stream of the product and/or service that is created, enhanced, or maintained. In an IT organization this includes the business and the IT sides. In a product company, it also includes marketing, sales, delivery, support, and development.

The Lean enterprise involves coordinating business, management, and delivery teams so that a sustainable stream of products can be delivered based on prioritized business need. Each area is important. Each area must be attended to, focusing on certain activities guided by Lean principles, as shown in Table 1.1, above. Like a three-legge
if any one area is neglected, the result is shaky.

Table 1.1 A Holistic View: Each Area Has a Part to Play

This area	Must attend to this work
Business	Continuously prioritize and decompose incremental needs across the organization Manage a portfolio of business needs Do release planning
Management	Organize cross-functional teams that can deliver incremental, end-to-end features Manage the value stream Bring visibility to impediments
Delivery Team	Work together, every day, and deliver fully tested and integrated code Learn how to deliver business needs incrementally Become proficient at acceptance test-driven development and refactoring

This book offers guidance on these focus areas to *anyone* in the enterprise who wants to learn how to transition to the Lean enterprise. This includes business people who want to understand Agile and technologists using Agile and trying to make it work. Whether or not you have experience with Agile, this book is a starting point for identifying how to apply the body of knowledge from Lean to the principles of Agile to make your enterprise better.

A QUICK REVIEW OF SOME LEAN PRINCIPLES

The foundation of Lean is based on several fundamental principles; these include

• Most errors are of a systemic nature and therefore your development system must be improved.

• You must respect your people in order to improve your system.

• Doing things too early causes waste. Do things just before you to do them: This is called "Just-In-Time," or JIT.

• Lean principles suggest focusing on shortening time-to-market by removing delays in the development process; using JIT methods to do this is more important than keeping everyone busy.

These are foundational in the sense that everything else comes from them. The first two form the cornerstone of W. Edwards Deming's work. Deming is the man generally credited by the Japanese with teaching them how to produce high-quality goods. Given that the Japanese did not always do this, it may be worth asking ourselves what the Japanese learned from this man—and what can we learn? Toyota added the JIT concept, which constitutes an essential component of Lean thinking.

Look to Your Systems for the Source of Your Errors

When something goes wrong, our normal tendency is to look for someone to blame. When a plane crashes we immediately ask ourselves, "Whose fault was it?" Was it the pilot's fault? (Blame *her*.) Was it the airline's fault? (Blame *them*.) Was it the manufacturer's fault? (Blame *them*.) Was some part on the plane faulty? (Blame *its* manufacturer.) But is that fair or at least is it sufficient? When we look for some*one* to blame, are we looking correctly? Perhaps it is the *situation* that the people found themselves in that caused—or at least contributed to—the problem.

Here is a typical example from software development. Say you are responsible for writing a feature of an existing system. You are given a document created by an analyst on the team that describes the functionality to be written. You are never afforded the opportunity to talk to someone who will actually be using the software, but rather must rely solely on this document. You write the code and, after testing, the new feature is shown to the customer, whereupon they declare, "This isn't what I asked for!"

Who would you blame? The *customer* for being unclear? The *analyst* for writing it up poorly? *You*, for not being able to follow the sp[illegible] *tester* for not testing it properly? With a little reflection, you ma[illegible] that no *person* is to blame; instead (or more often), the problem has to

do with the way they work together. In other words, in the current system, each person works separately in specialized roles. Feedback loops are nonexistent or inefficient and errors propagate. An Agile system would have the people work as a team. The customer, analyst, developer, and tester would talk among themselves to determine the customer's needs and how to best fulfill them. This is a better system. As errors occur, we look for ways to improve the communication process, continually driving down the number of errors that occur.

Improving communication is a major goal of Agile. Unfortunately, Agile practices tend to emphasize communication at the local level: among the team, between related teams, and with the customer. Agile offers only weak support for improving communication between teams that are only slightly related and practically none for communication up and down the value stream and across the enterprise. On the other hand, Lean practices promote communication in these larger contexts by focusing on the creation of end-to-end value, which provides a common context for everyone involved. This forces the different layers of the organization to communicate more frequently, with an emphasis on continuous process improvement, optimizing the whole, and delivering early and often. Lean thinking helps eliminate the delays that cause waste.

Respect People

Who should be involved in improving systems? Is this primarily a job for management or for the people doing the work? For Henry Ford, the answer was management. To him, management was much more intelligent than the workers and only they could be trusted to decide how to improve the manufacture of his cars. He had very little respect for the knowledge of workers.

The difficulty here is threefold. First, while Ford's policy allowed for setting up a very good static process for building one kind of car, it offered no flexibility. Recall Ford's famous statement that people could have "any color they wanted as long as it was black." Second, m processes are not static; they are always changing. Workers on t

will always understand the local conditions of a changing environment better than management because they have first-hand, local knowledge of what is going on—and that is what's needed to change processes on the line. Finally, Ford could get away with demeaning people in an age when a job's number-one value was providing a way to sustain one's family. Today, monetary compensation is no longer the overriding factor in selecting a job. That lack of respect would now translate into inability to keep quality employees. [1]

1 . Actually, Ford Motor Company was plagued by difficulties in keeping personnel and had to resort to high wages to do so. In their case, the time to train personnel was short because of how little knowledge one needed to work on the assembly line. This short training (often less than 15 minutes) would not be possible in the software-development world.

Respecting people—management and worker—allows flexibility in the process, continuous improvement of the process, and the ability to attract and retain the people qualified for the work.

In software development, respecting people includes the notion that the team doing the work is responsible for the process they follow. The process becomes their understanding of how to best develop software. When that changes, the process is changed. Hence, the process is the baseline by which the team builds software in the best way they know how within the constraints they are given.

MINIMIZING COMPLEXITY AND REWORK

A clear mantra for all developers is to minimize complexity and rework. Be clear: we are saying *minimize*, not eliminate. Although complexity and rework cannot be avoided entirely, the Lean principles can help reduce them from what has often been the norm.

Eliminating Waste and Deferring Commitment

Eliminating waste is the primary guideline for the Lean practitioner. Waste is code that is more complex than it needs to be. Waste occurs when defects are created. Waste is non-value-added effort required to create a product. Wherever there is waste, the Lean practitioner looks to the system to see how to eliminate it because it is likely that an error will continue to repeat itself, in one form or another, until we fix the system that contributed to it.

Deferring commitment means to make decisions at the right time, at the "last responsible moment": Don't make decisions too early, when you do not have all the information you need, and don't make them too late, when you risk incurring higher costs. Deferring commitment is a pro-active way to plan the process so that we either don't work on something until we need to or we set it up so that we can make decisions that can be reversed later when we get more information. This principle can be used to guide requirements, analysis, and system design and programming.

Deferring Commitment in Requirements and Analysis

We often think of commitment as an action or decision we make. But it can also be refer to time spent. Once we've spent time on doing something, it can't be undone—that is, we can't get the time back. In establishing requirements, we should ask, Where should I spend my time? Do I need to discuss all of the requirements with the customer? Clearly not. Some requirements are more important than others. Start with those requirements that involve functionality that is most important to the business as well as those that will create technical risk if they are not dealt with early.

The requirements that will be most important to the business are typically those that represent the greatest value to the customer. Agile methods handle this by directing us to delve into the requirements that customers feel are most important. This is one of the basic justifications for iterative development. But just looking at what features are important to customers is not a sufficient guide for selecting what to work on. We must also pay attention to archit

risk. Which requirements may cause problems if ignored? These are the ones that must be attended to.

Deferring Commitment in Design and Programming

Developers tend to take one of two approaches when forced to handle some design issue on which they are unclear. One approach is to do the simplest thing possible without doing anything to handle future requirements. [2] The other is to anticipate what may happen and build hooks into the system for those possibilities. Both of these approaches have different challenges. The first results in code that is hard to change. This happens because one does not consider the changeability of the code while writing it. The second results in code that is more complex than necessary. This occurs because, like most of us, developers have a hard time predicting the future. Thus, when they anticipate how to handle future needs, they often put in hooks (classes, methods, etc.) that actually aren't needed but that add complexity.

2. We are *not* referring to the eXtreme Programming mandate to do the simplest thing possible. That mandate is stated within the context of other actions. Unfortunately, that mandate is often misinterpreted as doing the simplest thing without attending to the need to handle future requirements.

An alternative approach to both of these is called "Emergent Design." Emergent Design in software incorporates three disciplines:

• Using the thought process of design patterns to create application architectures that are resilient and flexible

• Limiting the implementation of design patterns to only those features that are current

• Writing automated acceptance- and unit-tests before writing code, both to improve the thought process and to create a test harnes

Using design patterns makes the code easy to change. Limiting writing to what you currently need keeps code less complex. Automated testing both improves the design and makes it safe to change. These features of emergent design, taken together, allow you to defer the commitment of a particular implementation until you understand what you actually need to do.

Using Iterative Development to Minimize Complexity and Rework

The biggest causes of complexity are

• Writing code that isn't needed

• Writing code that is tightly coupled together

By doing iterative development, we avoid writing code that is not needed. That is, iterative development helps us discover what the customer really needs and helps us avoid building what isn't of value. Emergent design assists in the decoupling of "using code" to "used code" without adding unneeded complexity in the process.

Create Knowledge

Creating knowledge is an integral part of the Agile process. We build in stages so as to discover what the customer needs and then build it. By doing it this way we deliver value quickly and avoid building things of lesser (or no) value. We believe that software development is more a discovery process than a building process. By itself, software has little inherent value. Its value comes when it enables delivery of products and services. Therefore, it is more useful to think of software development as part of product development—the set of activities we use to discover and create products that meet the needs of customers while advancing the strategic goals of the company.

When viewed this way, it's clear that the role of software in IT organizations is to support the company's products and services. In software product companies, the software exists to support the and needs of the customers using it. Software is a means to an e the end is adding value for a customer—either directly, with a product,

or indirectly, by enabling a service the software supports. Software development should therefore be considered a part of product development.

You can look at product development as having three steps: [3]

3 . This is a gross over-simplification, of course.

1. Discover what the customer needs

2. Figure out how to build that

3. Build it

In software development, we seem to spend the most time talking about Step 3; however, the first two steps take the most time. Imagine having completed a software development project, only at the end to lose all of the source code. If you wanted to re-create the system as you had it, how long would it take? By re-create, we mean build it essentially the same way without trying to improve it; the only caveat is that you can leave out anything that is unnecessary. Most developers would say that this would take only 20 to 50 percent of the time it took to write it the first time. So, what were you doing the other 50 to 80 percent of the time? You were "discovering what the customer needs" and "figuring out how to build that."

Creating knowledge also means understanding the process that you use to build software to meet this discovered customer need. By understanding your methods, you can improve them more readily.

Deliver Early and Often

Another reason for doing iterative development is to deliver value to the customer quickly. This affords better market penetration, greater credibility of the business with the customer, strong loyalty, and other intangibles. However, it also provides for earlier revenue, allowing initial releases to pay for subsequent development.

This principle has often been called "deliver fast" but we feel it is better to think of it as "remove delays." Delays represent waste—if you remove them you will deliver faster. But the focus is on adding value to the customer without delay. Eliminating the delays results in going faster (from beginning to end). While the benefits of delivering fast are clear, it is essential that this is done in a sustainable manner.

Build Quality In

In order to sustain development speed, teams must build quality into both their process and their code. Building quality into their process allows a team to improve it by removing the waste it creates or requires. One way to do this is to define acceptance tests before writing code by bringing the customer, developer, and tester together. This improves the conversations that take place around the requirements and it helps the developers understand what functionality they need to write.

Building quality into code can also be achieved by using the methods described earlier to eliminate waste. Many developers spend much of their time discovering how to fix errors that have been reported. Without automated testing, errors creep in. Poor code quality and code that is hard to understand also contributes to wasted time.

Optimize the Whole

One of the big shifts to Lean thinking from a mass production mentality is discarding the belief that you need to optimize each step. Instead, to increase efficiency of the production process, look at optimizing the flow of value from the beginning of the production cycle to the end. In other words, getting each machine to work as efficiently as possible does not work as well as maximizing efficiency of the production flow in its entirety. Focus on the whole process – from the beginning (concept) to the end (consumption).

The problem with optimizing each step is that it creates large inventories between the steps. In the software world, these "inventories" represent partially done work (for example,

requirements completed, but not designed, coded, or tested). Lean proved that one-piece flow (that is, focusing on building an item in its entirety) is a much more efficient process than concentrating on building all of its parts faster. Inventories hide errors in the process. In the physical world, they may represent construction errors. In the software world, they may hide misunderstandings with the customer (requirements), or poor design (design documents), or bugs (coded but not tested code), or integration errors (coded, tested, but not integrated code) or any number of other things. The larger the inventory, the more likely there will be undetected errors.

FAST-FLEXIBLE-FLOW

A primary goal of Lean is to optimize the whole with speed and sustainability. This can be summarized as "fast-flexible-flow," which is the fundamental phrase used in Womack & Jones (2003). That is, get an idea into the development pipeline and out to the customer as fast as possible. Removing impediments to this flow is critical to improving the process.

This is very much the basis for the Agile practice of building one story at a time: getting it specified, designed, coded, and tested (done-done-done) by the end of the iteration. Scrum's Daily Stand-Up—stating, "Here's what I did, here's what I will do, here are my impediments"—is a direct reflection of this need to improve the process and remove anything that slows it down.

Focus on Time

Mass production looks at machine utilization. Lean focuses on time. In other words, instead of focusing on how well we are utilizing our resources, Lean says to reduce the time it takes to go from the idea to the value. Lean proposes that if we focus on going faster by improving our process, our costs will go down because we will have higher quality with fewer errors and less waste. Unfortunately, focusing directly on lowering costs does not necessarily have the benefit of improving quality or speed. In fact, it typically works the other way.

In the Lean world, we want to eliminate the waste of delays. Some common delays in software include

• The time from when a requirement is stated until it is verified as correct

• The time from when code is written until it is tested

• The time from when a developer asks a question of a customer or analyst until she gets an answer (the delay occurs here especially when e-mail must substitute for face-to-face contact)

Notice how these delays represent both risk and waste. This is because the waste that occurs if something goes wrong multiplies as the delays increase. Looking at these delays also illustrates why resource utilization is the wrong approach. Many individuals work on multiple projects simultaneously because they are often waiting for information or for other resources. For example, when a developer e-mails an analyst and then must wait for a response, she'll have another project to work on. Of course, this has everyone working on multiple projects, which further increases the delays in communication and other events. This doesn't even account for the added inefficiencies of context switching.

In manufacturing, Lean solves this problem by creating work cells that manage their own process and manage their work by pulling off a queue of prioritized tasks. In the software world, this is mirrored by creating self-directed teams that have all of the resources they need (for example, analysts, developers, and testers) and pulling from a product backlog.

REFLECTIONS ON JUST-IN-TIME (JIT)

There is a parallel between standard manufacturing and JIT in the same way there is a parallel between the Waterfall software model and iterative development. In Waterfall, we take all of our "raw materials" (requirements) and start processing them in big batches. Resources required by the team (DBAs, analysts, testers) are available at different

intervals, so we batch-process our work (analysis, design, code, test) to fully utilize them. Since we are building so many things at once, we make a great effort to figure out what needs to be done before starting the manufacturing process (akin to the heavy analysis of Waterfall). In JIT we work only on those parts that we need to, and then, only just before they are needed. This is akin to taking a story in almost any Agile method and doing the analysis on it just before it is built and validated.

By performing work in small, complete steps, JIT in the software arena gives us the ability to change direction at the end of each small completion—without any wasted effort. One of the mantras of Lean manufacturing is to minimize work-in-process (WIP). Agile methods strive for that as well.

Of course, accomplishing JIT is not easy. It requires a smooth, low error-rate process. This requirement, however, makes deficiencies in the process more evident: It makes them easier to see and therefore easier to fix. This parallels the cornerstone of Scrum's efforts to remove impediments to the one-piece flow of stories.

JIT has other advantages. It not only uncovers problems in the process, it also exposes problems in production before they have too much of an impact. In mass production, errors are often discovered only in the later stages of production. If there is a large inventory between steps, a lot of bad inventory may be produced before the error is detected. In software, a delay in discovering the error in a requirement, for example, will result in wasted effort to build and test that piece. It also adds complexity to the system even though it doesn't provide value to the customer (because unneeded features are typically never removed). Essentially, if we can deploy (or at least demonstrate) finished code to customers quickly and in small pieces, we can get their feedback and know if we are producing something of value.

Thus, JIT provides guidance for software development. We can look at Agile methods as an implementation of JIT principles. We don't do analysis of a story until just before building it. We analyze, design,

code, and test just before each stage is needed, which should reveal impediments to our process. JIT encourages us to build things in smaller batches, and it provides the basis for quick feedback from the customer.

Figure 1.1 represents the Waterfall model as a series of steps that take input from the prior step and hand it to the next step.

Figure 1.1 Waterfall as mass production

Table 1.2 compares the hidden costs in manufacturing as compared with the equivalent costs in software. Note, however, that in software, the costs are usually greater because "inventory" in software degrades much faster than most inventory in manufacturing.

Table 1.2 Comparing the Costs and Risks in Mass Manufacturing and Waterfall Software Development

	Manufacturing Mass Production	Waterfall Model
Hidden Costs	Transportation	Handoffs
	Managing inventory and storage	Lots of open items; can lead to overwhelming the workforce
	Capital costs of inventory	Cost of training people to build software
Risks	Building things you don't need because production goes on after needs go away	Building things you don't need because requirements aren't clear or customers change their minds
	Inventory becoming obsolete	Knowledge degrading quickly If a line is discontinued, all WIP wasted
	Huge latency if an error occurs	Errors in requirements discovered late in the process Errors in completed code discovered late in testing

VALUE STREAM MAPPING

The value stream is the set of actions that takes place to add value for a customer from the initial request to the delivery of the value. The value stream begins with the initial concept, moves through various stages to one or more development teams (where Agile methods begin), and on through to final delivery.

The value stream map is a Lean tool that practitioners use to analyze the value stream. Value stream mapping involves drawing pictures of the process streams and then using them to look for waste. The focus is on improving the total time from beginning to end of the entire stream while maintaining this speed in the future (that is, you cannot take shortcuts now at the expense of future development).

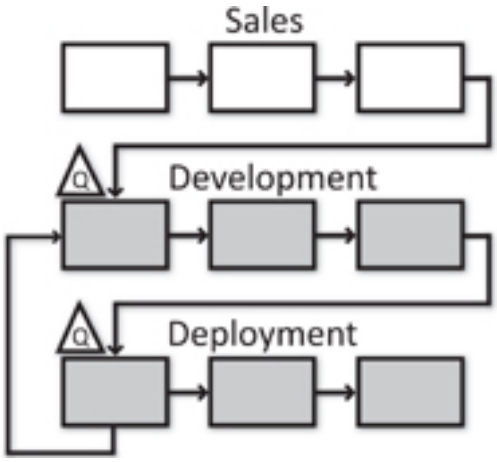
One of the great benefits of value stream mapping is that it shows the entire picture. Many Agile practitioners focus on improving the team's performance. Unfortunately, in many cases, the team is not the cause of the development problems—even when it looks that way. Value stream mapping shows how to "optimize the whole" by identifying waste: delays, multi-tasking, overloaded people, rework, late detection of problems, and so on, which affects quality and slows down delivery.

Using Value Stream Mapping to Get to True Root Cause

At one of our Lean Software Development courses, we had two students from a medium-sized company's development team. It was clear to the company that they had problems resulting from poor code quality which, therefore, was an issue for the development team. In a nutshell, when the company's products were installed at their customers' sites, they often had issues that needed to be fixed, which slowed down new development. They came to us for help because they could no longer just hire more developers to fix the problems.

Near the start of the course, students create an "as-is" value stream map. The map that they drew for their organization is shown in Figure 1.2.

Figure 1.2 As-Is value stream map



The loopback shown in the figure occurs when a customer has a problem that requires work to go back through development. The queues (shown with triangles) indicate that work was often in a wait state, both for development and for deployment. The loopback was particularly disruptive because development teams would start work on the next project only to be pulled off to work on the previous system that had gone awry.

In some sense, the value stream map presented little new information. It illustrated what was known: Developers had a quality problem and were having to do a lot of rework. But the value stream map presented a new perspective. First, it showed the entire development process (including the marketing aspect). Second, a value stream map suggested geting to the root cause of the problem, which was system failure at the customer site. That is, we use value stream maps to identify problems and we use other Lean thinking to get to root cause.

For root cause analysis, it is common in Lean to use the "Five Whys." This technique, credited to Sakichi Toyoda, the founder of Toyota Industries, involves asking why something happened and then why that happened and then why that happened, continuously exploring the cause-and-effect relationships underlying a particular problem until it drills down to the root cause.

In our students' case, the technique started with the question "W
we having to rework the system?"

A: Because the programs do not function properly on our customers' servers.

Q: Why do the programs not function properly on our customers' servers?

A: Because the code was designed one way, but the servers are configured for another way.

Q: Why are our customers' servers being configured differently from how it was expected?

A: Because our customers are not following our guidelines for server configuration.

Q: Why are our customers not following our guidelines for server configuration?

A: Because they aren't aware of the guidelines.

Q: Why aren't these customers aware of them?

A: Because sales, who is supposed to make sure they know of this configuration requirement, isn't telling them.

Q: Why isn't sales telling our customers they need to do this?

A: Because when a customer is ready to buy, sales tends to shut up and just get the contract signed. Closing the deal seems to be the most important thing to sales.

This series of questions illustrates many points. First, it's not really always *five* whys. Five is often enough, but sometimes you have to keep questioning until you get to the root cause. In this case, sales was not informing the customers that the machines needed to be configured a particular way. Second, the problem may not be wh think it is. In this case, the assumption was that code quality wa problem; in fact, the problem was that the code was not flexible

enough to run on misconfigured servers. Either the code could be fixed or the servers could be configured properly. Third, the origin of the problem is not always where you think it is. In this case, the company was fairly sure that this was a development team problem (which is why two people from the development organization were there), when in fact, the problem lay in sales department.

This highlights a critical failure in many Agile approaches that focus on the team: *The problem may not be with the team even though many Agilists say to start there*. A value stream map enables us to see the whole picture and to question our assumptions.

Toward the end of the course, these two participants then created their "to-be" value stream map. That is, a map of their value stream as it should be done. This new value stream map required that their customers were aware of running configuration checks.

The conversation about this change grew quite animated. There was fear that sales would not like this new requirement. After all, here is a customer ready to pay money, but before they can pay, they have to do some additional upfront work. This would seem contrary to sales' desire to close the deal as quickly as possible. Probably, they would not like any perceived impediment to the deal.

But the value stream analysts looked at the bigger picture. They focused on how the customer would react. They felt that most customers would see that the company was acting responsibly and putting the customer's interests first. And if they lost a few customers who did not want to make that upfront commitment, then that was OK. By having all customers either follow the new process or cease to be customers, the organization would remove the bottleneck and be able to deliver more value more quickly. They saw that their problem lay not in having enough customers; it was the waste in their process. The fact was they had more customers than they could support.

It also illustrates how metrics and performance awards can be counterproductive. Sales people were being rewarded on system's total

The rewards should have been based on systems *installed*. Metrics and rewards that focus on only part of the value stream are often counterproductive.

Perhaps most interesting is that this company experienced a significant team performance improvement without changing what the team was doing at all.

The Results

Implementing this "to-be" value stream map resulted in significant conversations across the organization. Everyone, including sales, learned and started to see benefits. Development was pleased not to have to make unnecessary changes to their approach.

A few months later, the company did a second round of value stream mapping. They started with the previous "to-be" map. Armed with a better understanding of their process, they applied the Lean principle to defer commitment as long as practical and to move significant server analysis downstream, doing it just in time for development. They could see exactly where to do this to maximize the needs of development without unnecessarily hampering sales. This reduced delay in their work stream even more! It is also interesting to note that they didn't lose any customers. They cleverly presented the "requirement" of pre-configuring the systems as a service the company offered the customer as the step just before installation.

LEAN GOES BEYOND AGILE

By providing time-tested principles, Lean provides guidance for our Agile practices when we are in new situations. Lean tells us to focus on time of development, not resources utilized. Lean reminds us to optimize the whole instead of trying to get each step done as efficiently as possible.

Lean takes us beyond Agile's often myopic focus on project/team/software. To attend to the entire enterprise, we ne look at how products are selected for enhancement and how tea work within the structure of the organization. Agile does not help here

but is often severely and adversely affected by the lack of good structures within which the teams must work.

SUMMARY

Lean tells us to focus on improving the system we use to produce software by focusing on processes that support the team. Since the team knows more about how they develop software than anyone else, they need to create and improve these processes.

Lean provides seven principles for software development:

• Respect people

• Eliminate waste

• Defer commitment

• Create knowledge

• Deliver fast

• Build quality in

• Optimize the whole

A fundamental goal for Lean is fast-flexible-flow. That is, it is useful to think of the development process as a pipeline where production takes place. Anything that slows down the pipeline causes waste. In software, waste includes delays, errors, misunderstandings, and waiting for resources. By removing impediments to this flow, we improve our process.

Value stream mapping is a vital tool for analyzing process in order to reduce delay and waste.

Thus, Lean provides guidance for Agile teams. In fact, Scrum can be seen as a manifestation of Lean principles. Understanding Lean assist in implementing Scrum. Lean also can be applied through

the enterprise, thereby assisting to implement Scrum throughout the enterprise.

TRY THIS

These exercises are best done as a conversation with someone in your organization. After each exercise, ask each other if there are any actions either of you can take to improve your situation.

• In your organization, how long does it take for an idea to get from business charter (the document used to build the business case and a vision for justifying the project) to delivered capability (months, years)? This is considered the cycle time, which any new system should minimize.

• Does batching ideas into projects increase or decrease this cycle time?

• What are impediments to smaller, more frequent deliveries in your organization?

• How can you influence your organization to make visible the costs due to delays?

RECOMMENDED READING

The following works offer helpful insights into the topics of this chapter.

Bain. 2008. *Emergent Design: The Evolutionary Nature of Professional Software Development*. Boston: Addison-Wesley.

Kennedy. 2003. *Product Development for the Lean Enterprise: Why Toyota's System Is Four Times More Productive and How You Can Implement It*. Richmond, VA: Oaklea Press.

Poppendieck and Poppendieck. 2006. *Implementing Lean Software Development: From Concept to Cash*. Boston: Addison-Wesley