Theory of Constraints, Lean, and Agile Software Development

WRITTEN BY AMR ELSSAMADISY AND JOHN MUFARRIGE

THURSDAY, 08 MARCH 2007

Delivering More Business Value Where Needed

Within the software development community, one of the biggest movements over the past decade has been Agile Development whereby teams adopt practices and attitudes consistent with the now famous Agile Manifesto. Additionally, there has been much discussion over the past four to five years about applying principles from the Theory of Constraints (ToC) and Lean Product Development (Lean) to software development. This has had a tendency to muddy the surrounding waters as teams question whether they should apply Agile, ToC, or Lean concepts. Are these three approaches mutually exclusive? Is there some hidden magic that can be unlocked by careful application of all three? Isn't it hard enough just trying to be Agile, without also trying to be Lean and ToC-ish? In this article we give an overview of Lean and ToC and show how they can be used in conjunction with Agile practices to focus on an organization's business value. By using elements of Lean, ToC, and Agile together more business value can be delivered with less effort.

Introduction

What is important to your organization? The answer to that question is as varied as the number of organizations that exist. Software that we build should always address/advance one or more of the organization's goals. With advent of Agile development we have become much more effective at meeting those goals. We have also become adept at recognizing and responding to changes and making course adjustments appropriately. So is that enough? Are we there yet?

There is more that can be done. By borrowing some good ideas from the business world, namely ToC and Lean, we can target our customers' needs and goals even better. In this article we briefly outline how to do so and give references for further reading.

The diagram in Figure 1 shows a very simple process that will be used for illustration of Lean and ToC below. Each step has a speed indicated in parentheses below the step name. Inventory builds up between two steps when they are of mismatched speed - therefore because step A is faster than step B, an excess product of A waits to be processed by B.

Simple Sequential Process
(SSP)

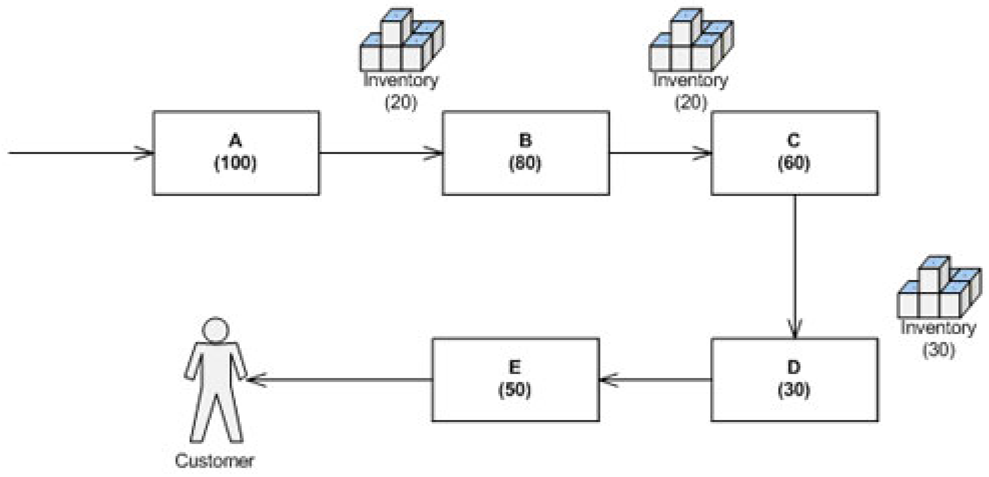

Figure 1: Simple Sequential Process (SSP)

Lean

Much has been written about Lean Manufacturing and its application to software development. Lean is all about defining value as the sole service/product delivered to the customer. Everything else that does not either directly or indirectly provide value to the customer is seen as waste and is to be eliminated. Broadly, the steps in Lean, as defined by Womack and Jones [2] are:

1. Define value for your customer. Everything else is waste. Learn to see that waste.
2. Eliminate all waste. Any step that does not provide value for the customer should be a candidate for modification or removal from the process.
3. Make steps in your process flow. Each step in a process should directly feed into the next one without delay. Inventory is one form of waste because it does not deliver value to the customer.
4. Pull work. That is, a step in the process should not create anything until it is needed by the step after it. This brings inventory levels down to zero and only processes work 'pulled' by the customer at the end of the process chain.
5. Work towards perfection. That is, go back to step one and make things better - remove more waste.

With respect to the SSP process in Figure 1, value would be defined with respect to our customers - anything they are willing to pay for (not intermediate results). We would next see everything that does not deliver value as a 'waste' and work diligently to remove that waste until the steps flow from one to the other without waiting or inventory. We would then only create product when the customer (at the end of the process) requests - or pulls - it.

By continuously eliminating waste the entire system delivers more and faster with less effort. Mary and Tom Poppendieck [7, 8] have given us a guide describing how this relates to software development directly. They help us see different forms of waste in software development such as stale requirements, gold plated code, and functions that are rarely (if ever) used. They also give us some strategies to help eliminate them.
So all is well and good. We're done! Well... Lean doesn't address where to start eliminating waste. It turns out that where you start eliminating waste is an important factor. You can continue to eliminate waste for a long time until you get to 'flow' and a 'pull' system.

Theory of Constraints
The Theory of Constraints, created by Eliyahu Goldratt and introduced to the world in the business novel "The Goal," [4] simply asserts that there is only one bottleneck in a system of production. Throughput of the overall system will always be limited by that bottleneck. The idea is that if you want to increase the overall throughput of the system you need to focus almost exclusively on improving the performance of the bottleneck (also known as the "constraint," thus the "Theory" of Constraints). Efforts spent elsewhere are waste (in Lean terms) and could even be counterproductive.

So what does ToC advise we do? We should:

1. Find the bottleneck (the slowest point in the system).

2. Exploit and protect the bottleneck so that it does not get interrupted. Protecting the bottleneck is usually done with inventory, so if a step upstream of the bottleneck goes down, the bottleneck can continue to function.

3. Make all other steps in the process subservient to the bottleneck's capacity - in English, "slow them down" and prioritize any work that affects the bottleneck.

4. Elevate/improve the capacity of the bottleneck until it is no longer the slowest step in the system.

5. Go to step 1 and continue.

In Figure 1, the bottleneck is step D running at a speed of 30. We would keep the inventory between C and D, slow all steps to a speed of 30 (thus removing all other inventory), and use the resources to make D go faster until it is no longer the slowest step in the system.

For this to work, the bottleneck must be a global bottleneck (for the entire system) and not a local one (for a single step in the system). Otherwise, addressing a local bottleneck will make a non-bottleneck step faster and will be effort down-the-drain because the real bottleneck has not been addressed. In our example, let's assume that the group responsible for step C have their own internal process and they work on making this nested process better. They will find a bottleneck for step C and remove it and therefore step C will increase to a speed greater than 60. This is great for step C, but actually does nothing to improve the system as a whole. In fact, it will probably result in increased inventory waiting to be processed between steps C and D.

ToC is very specific of where to start: the BOTTLENECK. The location of that bottleneck differs from organization to organization, so you will have to find your own bottleneck. Fortunately, they are usually not that hard to find - one of the biggest indicators is a pile of inventory stacking up before the slowest step as it gradually gets farther and farther behind the rest of the system.Specifically, in software development, the "inventory" that piles up is composed of requirements waiting to be processed (this can be development, testing, integration, deployment, etc...). Typical bottlenecks include requirements specification (the rest of the system is starved for requirements because it takes so long to create requirements documents), system test/ QA, and defects (rework inventory). We find it interesting that actual coding work is commonly not the bottleneck, and yet it seems to receive a disproportionate amount of attention. Maybe coders are the biggest whiners?

What's interesting about this approach is that it is completely contrary to the notion of improving "personal productivity" or efficiency. ToC plainly states that not only is it ok but many times it's actually desirable for parts of your system to operate below their maximum capacity. To put this in terms of a software development team, let's say that UI QA has become your bottleneck, meaning that you have a buildup of user interface requirements or features waiting to be tested by QA. As the rest of the team continues to define, design, and implement new requirements, the amount of "inventory" waiting to get processed by QA will continue to grow because it is the bottleneck. So step 3 of ToC says you should slow down all non-bottleneck steps to operate at the same pace of the bottleneck, meaning your requirements gathering, design, and development should all be restricted to the extent that you are releasing implemented features to QA at a slow enough pace that QA can immediately start testing them. So now, if your team size has not changed, it's entirely likely that your QA team is working at or near its capacity while the rest of the team is working below its capacity; literally they might be sitting around waiting for work to do! So what do you do in this situation if you don't like the notion of paying a bunch of analysts and developers to sit around and do nothing or only work part time? According to ToC, you should either scale back the number of people gathering and implementing requirements (if you don't want them to be idle) or you can increase the capacity of your bottleneck (step 4). In this case you can increase the number of features QA can test at any given time or reduce the cycle time per test (help QA figure out to how test more quickly). Either of these steps will increase the capacity of QA, meaning the other parts of the team can scale up again until another bottleneck is revealed (step 5).

Agile
Lean and ToC help identify and target waste to improve to our organization. They do not tell us how to do so although there have been efforts in the field to apply both Lean and ToC to software development (Poppendiecks' two books [7, 8], and Agile Management for Software Engineering [9] by David J. Anderson come to mind).

Agile development is all about practicing a better form of building software. There are methodologies that address ways to build software with several practices together such as Scrum, XP, Crystal, and others. The practices are useful in and of themselves and can be used as surgical tools as long as we are cognizant of how a development practice affects the software development system.

To adopt the correct agile development practices we must be aware how each practice helps eliminate waste in one or more steps of a development cycle. So, for example, test driven development helps reduce the pile (inventory) of bugs waiting to be fixed. Pair programming reduces ‘hand-offs' between developers with different expertise to build a cross-functional requirement. Cross-functional teams help eliminate the waste from hand-offs and pending work as it moves from team to team.

Putting It All Together: Targeting with ToC, Eliminating Waste with Lean, and Elevating the Constraint with Agile Practices

Lean and Agile aren't very good about indicating where in our system to start improving, but ToC is. To maximize the business value (to be read "most directly address our organization's goals ") we start with step 1 in ToC:

1. Find the bottleneck.

Then, even though inventory is a form of waste, in this case it is ‘necessary waste' because it keeps the system from getting even slower if something happens to the steps that feed the bottleneck. With that we have done step 2 of ToC:

2. Exploit and protect the bottleneck.

The final step we will use from ToC before switching to Lean is to slow down the entire system so that steps run no faster than the bottleneck.

3. Make all other steps in the process subservient to the bottleneck's capacity (and thus achieve flow).

Anything running faster than the bottleneck does not make the system run faster and is a form of waste. Once we bring down the speed we have done two things: we have released excess resources from non-bottleneck steps that we can use elsewhere, and we have just found ourselves at step 3 in Lean - the system is ‘flowing'!

This is the perfect hand-off position to Lean's step 4:

4. Pull work.

Now our system does not do any extra work until it is needed and it is running at the bottleneck's speed. At this point Lean tells us to continue removing waste and ToC tells us to increase the capacity of the bottleneck until it is no longer the bottleneck. ToC is the more specific of the recommendations:

5. Elevate/improve the capacity of the bottleneck until it is no longer the bottleneck (using Agile practices).
So how do we elevate the bottleneck? How do we increase its capacity? You guessed it - this is where the Agile practices come in. We now target the Agile practices to surgically improve the process. We use the resources that were freed up earlier in step 3. We don't have a large collection of practices to adopt. For this to work well we must be aware of each Agile practice and what parts of the organization it addresses and improves.

Conclusion

We have given an overview of both Lean and ToC and proposed a way to use them alongside Agile practices to more effectively target your organization's business goals. This article is meant to pique your curiosity and whet your appetite for Lean and ToC. If we have been successful in doing that, then go read more with our advice in mind.

References

These references are good starting points to the topics of Lean, ToC, and Agile practices that leverage either/both of these methods.

Lean Production

1. Womack, Jones, and Roose, The Machine that Changed the World, Harper Perenial, 1991.
2. Womack and Jones, Lean Thinking, Free Press, 2003.
3. Liker, The Toyota Way, McGraw Hill, 2003.

Theory of Constraints

4. Goldratt and Cox, The Goal, 3rd edition, North River Press, 2004.
5. Goldratt, Beyond the Goal: Eliyahu Goldratt Speaks on the Theory of Constraints (Audio), Coach Series, 2005.
6. Scheinkopf, Thinking for a Change: Putting TOC Thinking Processes to Use, CRC, 1999.

Agile Software Development

7. Poppendieck and Poppendieck, Lean Software Development: An Agile Toolkit for Software Development Managers, Addison Wesley Professional, 2003.
8. Poppendieck and Poppendieck, Implementing Lean Software Development: From Concept to Cash, Addison Wesley Professional, 2003.
9. Anderson, Agile Management for Software Engineering: Applying Theory of Constraints for Business Results, Prentice Hall PTR, 2003.

About the Authors

Amr Elssamadisy is a Principal Consultant with Valtech, where he helps clients build better software using the latest technologies and, of course, adopting and adapting Agile practices. As a software practitioner, he plays multiple roles on different teams including coach, instructor, developer, architect, tech-lead, Scrum master, and project manager. He is also author of Patterns of Agile Practice Adoption: The Technical Cluster which guides the reader in crafting and implementing an Agile adoption strategy.

John Mufarrige is a Senior Consultant with Valtech, where he holds down odd jobs as Java/J2EE developer, instructor, Agile development mentor, and general troublemaker. His other work-related interests include testing, metrics, and team dynamics. He started his professional career in the mid-nineties as a VB/ASP developer, then saw the light in 2000 and shifted wholeheartedly into the Java realm.